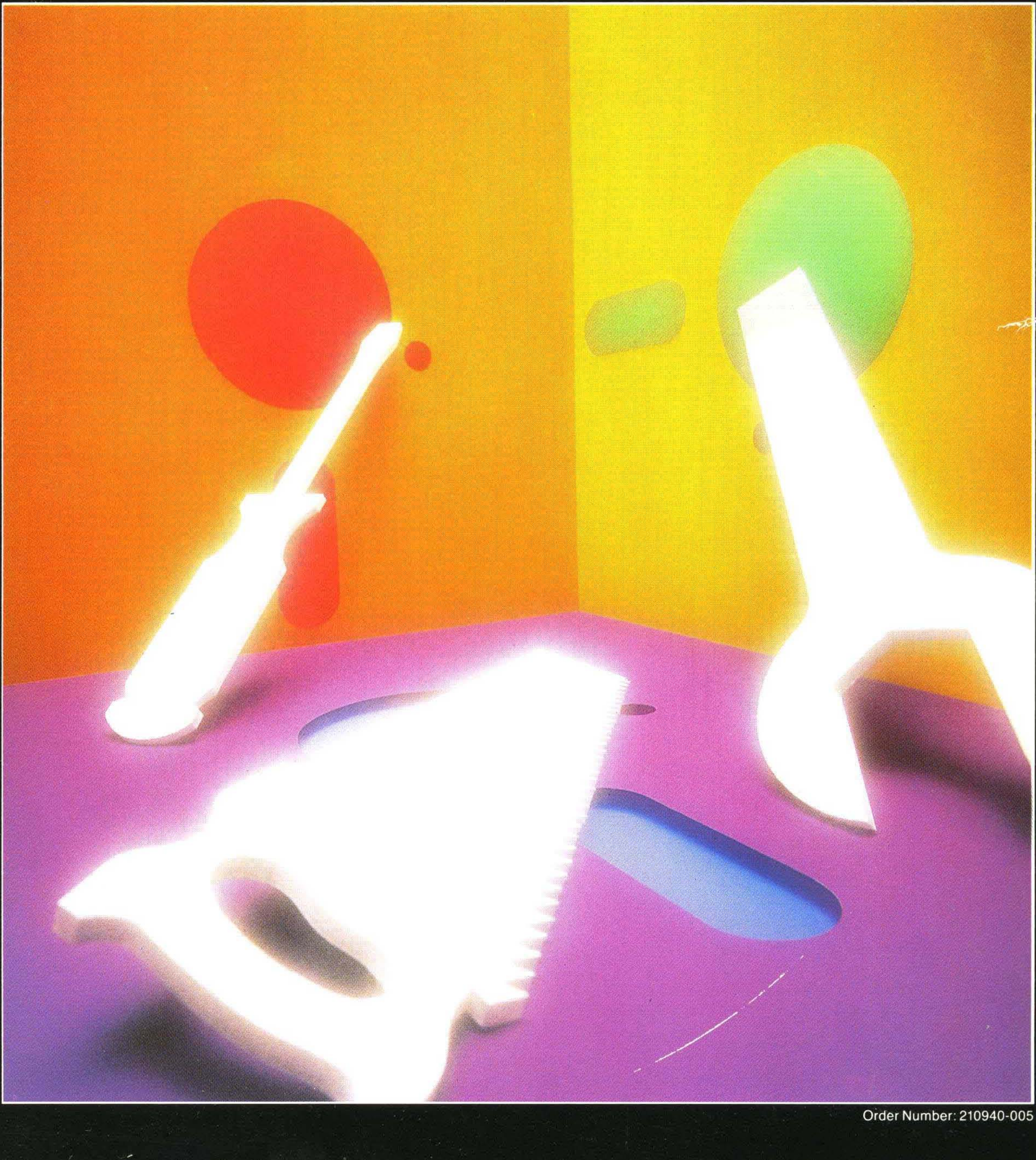
intel®

Development Tools Handbook

Software, Tools and Systems



Order Number: 210940-005

LITERATURE

To order Intel literature write or call:

Intel Literature Sales P.O. Box 58130 Santa Clara, CA 95052-8130	Intel Literature: (800) 548-4725

Use the order blank on the facing page or call our **Toll Free** Number listed above to order literature. Remember to add your local sales tax and a 10% postage charge for U.S. and Canada customers, 20% for outside U.S. customers.

1987 HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information.

NAME	ORDER NUMBER	*PRICE IN U.S. DOLLARS
COMPLETE SET OF 9 HANDBOOKS Save $50.00 off the retail price of $175.00	231003	**$125.00**
MEMORY COMPONENTS HANDBOOK	210830	**$18.00**
MICROCOMMUNICATIONS HANDBOOK	231658	**$20.00**
EMBEDDED CONTROLLER HANDBOOK (includes Microcontrollers and 8085, 80186, 80188)	210918	**$18.00**
MICROPROCESSOR AND PERIPHERAL HANDBOOK (2 Volume Set)	230843	**$25.00**
DEVELOPMENT TOOLS HANDBOOK	210940	**$18.00**
DOS DEVELOPMENT SOFTWARE CATALOG	280199	N/C
OEM BOARDS AND SYSTEMS HANDBOOK	280407	**$18.00**
MILITARY HANDBOOK	210461	**$18.00**
COMPONENTS QUALITY/RELIABILITY HANDBOOK	210997	**$20.00**
SYSTEMS QUALITY/RELIABILITY HANDBOOK	231762	**$20.00**
PRODUCT GUIDE Overview of Intel's complete product lines	210846	N/C
LITERATURE PRICE LIST List of Intel Literature	210620	N/C
INTEL PACKAGING OUTLINES AND DIMENSIONS Packaging types, number of leads, etc.	231369	N/C

*These prices are for the U.S. and Canada only. In Europe and other international locations, please contact your local Intel Sales Office or Distributor for literature prices.

intel

LITERATURE SALES ORDER FORM

NAME: ____________________

COMPANY: ____________________

ADDRESS: ____________________

CITY: ____________________ STATE: __________ ZIP: __________

COUNTRY: ____________________

PHONE NO.: (   ) ____________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal __________

Must Add Your Local Sales Tax __________

Must add appropriate postage to subtotal (10% U.S. and Canada, 20% all other) → Postage __________

Total __________

Pay by Visa, MasterCard, American Express, Check, Money Order, or company purchase order payable to Intel Literature Sales. Allow 2-4 weeks for delivery.

☐ Visa ☐ MasterCard ☐ American Express Expiration Date __________

Account No. ____________________

Signature: ____________________

Mail To: Intel Literature Sales
P.O. Box 58130
Santa Clara, CA
95052-8130

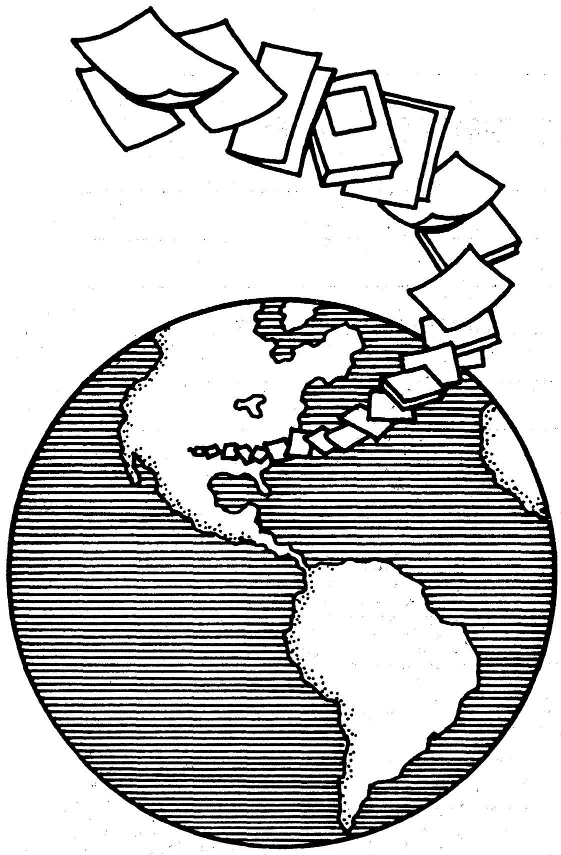
International Customers outside the U.S. and Canada should contact their local Intel Sales Office or Distributor listed in the back of most Intel literature.

European Literature Order Form in back of book.

Call Toll Free: (800) 548-4725 for phone orders

Prices good until 12/31/87.

Source HB



We Bring Our World to Your Door

Intel's New Product Literature Subscription Service.

Keeping up with today's technology takes a lot of time and effort. With Intel's new Literature Subscription Service you will receive a package of current literature plus automatic quarterly updates on all the latest product and service news from Intel. From microprocessors — to peripherals and memories — to OEM boards, systems and software, you can choose to receive information from one, or all three, product categories for an entire year at a low one-time cost.

Save Time and Money. Subscribe Today.

Save 10% when ordering two or more packages.

To order, use the literature order form provided in this book or call TOLL FREE 800-548-4725

The charge for this service covers our printing, postage and handling costs only.

intel®

Each Literature Package Contains:

Newly published Data Sheets, Fact Sheets, Application Notes, Reliability Reports, Errata Reports, Article Reprints, Promotional Offers, Brochures, Flyers, Benchmark Reports, Technical Papers and more...

In Addition, Each Individual Package Contains:

1

Microprocessors, etc.

Product Line Handbooks on Microprocessors, Development Tools and Embedded Controllers.
plus
Quality/Reliability Information, The Product Guide, Literature Guide and Packaging Information.
plus
Three Quarterly Updates containing all new documentation on these products. (Retail Value of Handbooks alone: $81)

Your price for the complete package with quarterly updates: $70

Order Number 555100

2

Peripherals, Memories, etc.

Product Line Handbooks on Peripherals, Microcommunications, Memories and EPLD
plus
Quality/Reliability Information, The Product Guide, The Literature Guide, Packaging Information and other supporting information.
plus
Three Quarterly Updates containing all new documentation on these products. (Retail value of Handbooks alone: $83)

Your price for the complete package with quarterly updates: $70

Order Number 555101

3

OEM Boards, Systems and Software

Produce Line Handbooks on OEM Boards and Systems
plus
Quality/Reliability Information, The Product Guide, The Literature Guide, and other supporting information.
plus
Three Quarterly Updates containing all new documentation on these products. (Retail Value of Handbooks alone: $38)

Your price for the complete package with quarterly updates: $60

Order Number 555102

Customers outside the U.S. and Canada should order directly from the U.S. on the U.S. literature order form.

Offer expires 12/31/87

FREE

DEVELOPMENT SOFTWARE CATALOG

Intel's *DEVELOPMENT SOFTWARE CATALOG* contains a complete description of Intel's high level languages, utilities, assembly languages, editors and debuggers running on DOS, VMS, ISIS, and iNDX.

Call or write today for your *FREE COPY.*

Call TOLL-FREE 1-800-87-INTEL for your free copy, or fill out the coupon below:

Clip and mail to:

Intel Corporation
P.O. Box 58065
Santa Clara, CA 95052-8065

(H89) ☐ *YES!* I want my free copy of Intel's Development Software Catalog.

☐ Have an Intel Sales Representative Call Me.

Name ______________________ Title __________
Company ____________________ Mailstop __________
Address ________________________________
Phone (________) ________________________
City ________________ State ________ ZIP ________

DTO-355B-BR-CP Expires 12/87

intel®

DEVELOPMENT TOOLS HANDBOOK

1987

About Our Cover:
Intel's open environment provides developers with a portable set of dedicated tools with networking capabilities resulting in greater productivity, expandability, upgradeability and a shorter time to market. Our powerful, dedicated and proven set of software development tools are abstractly represented here as another familiar set of dedicated and proven tools.
Concept/Design: Hall Kelley, Concept/Photography: R. J. Muna

Additional copies of this manual or other Intel literature may be obtained from:

Intel Literature Department
SC6-59
P.O. Box 58065
Santa Clara, CA 95052-8065

Table of Contents

Alphanumeric Index ix
Intel's Development Environment—The Complete Solution xi

CHAPTER 1

Microcomputer Development Languages

DATA SHEETS

386 Software Tools 1-1
80286 Software Development Package 1-14
PASCAL 286 1-19
PL/M 286 1-22
iC 286 1-26
8086/8088 Software Development Package for Series II/PDS 1-30
8086/8088/80186/80188 Software Packages 1-40
FORTRAN 80 8080/8085 ANS FORTRAN 77 Intellec® Resident Compiler 1-61
Pascal 80 Software Package 1-65
PL/M 80 High-Level Programming Language 1-70
8087 Support Library 1-73
80287 Support Library 1-77
X 8051 Software Packages 1-81
MCS®-48 Diskette Based Software Package 1-90
X MCS®-96 Software Development Packages 1-92
VAX*/VMS* Resident Software Development Packages for 80286 1-100
VAX*/VMS* Resident 8086/8088/80186 Software Development Packages 1-106

PRODUCT BRIEFS

Ada 286 Compilation System 1-113

FACT SHEETS

Intel Microprocessor Languages 1-115

ARTICLE REPRINTS

AR-59 Modular Programming in PL/M 1-117
AR-136 PL/M 86 Combines Hardware Access with High-Level Language Features 1-124
AR-200 Compiler Organization Techniques 1-131
AR-239 PL/M 51: A High Level Language for the 8051 Microcontroller Family 1-137
AR-377 Integrated Tools Accelerate Code Development 1-144
AR-388 Ada Task Synchronization in a Multiprocessor System with Shared Memory 1-150
AR-390 Ada Capabilities for Today's Microprocessors 1-161
AR-501 .OBJ Lessons 1-165

CHAPTER 2

Microcomputer Software Development Tools

DATA SHEETS

PSCOPE Monitor 386ES (P-MON 386ES) 2-1
PSCOPE-86 for DOS High-Level Application Program Debugger 2-5
PSCOPE High-Level Program Debugger for iRMX™, XENIX*, Series III and Series IV 2-12
Debug MONitor 386 (D-MON 386P) 2-20
8086 Handyman 2-23
AEDIT Source Code and Text Editor for PC-DOS 2-28
iPAT™ Performance Analysis Tool 2-31

PRODUCT BRIEFS

80386 Development Environment 2-44

APPLICATION NOTES

AP-243 Debugging with Intel on the VAX* 2-46

Table of Contents (Continued)

AP-253 Adding Value to Intel's NDS II Development System Network with CP/M-80 2-56

ARTICLE REPRINTS

AR-225 Debugging Catches up with High-Level Programming 2-64

AR-319 Software Development 2-70

AR-352 Integrated Environment Speeds System Development 2-75

CHAPTER 3

In Circuit Emulators

DATA SHEETS

iSBE-96 Development Kit Single Board Emulator 3-1

I2ICE Integrated Instrumentation and In-Circuit Emulation System 3-9

VLSiCE™-96P In-Circuit Emulator for the MCS®-96 Family of Microcontrollers 3-62

ICE™-386 In-Circuit Emulator for the 80386 3-73

ICE™-5100/044 In-Circuit Emulator for the RUPI™-44 Family 3-83

ICE™-5100/252 In-Circuit Emulator for MCS®-51 Family of Microcontrollers 3-91

ICE™-5100/452 In-Circuit Emulator for the UPI™ 452 Family of Programmable I/O Processors 3-101

APPLICATION NOTES

AP-239 Customer Applications of the EMV-88 Emulation Vehicle 3-110

AP-262 Using Procedures to Speed I2ICE™ System Debugging 3-123

AP-273 Developing MCS®-96 Applications Using the SBE-96 3-174

CHAPTER 4

Network Development Systems

DATA SHEETS

OpenNET™ Network Resource Manager (NRM) iMDX 460 4-1

Compilengine iMDX 485CE 4-6

OpenNET™ Personal Computer Link 4-11

NDS II/VAX* Link Networking Software 4-17

iMDX 555 NDS-II NRM OpenNET™ Upgrade 4-20

iMDX 581 ISIS Cluster Board Packages 4-24

Intel Asynchronous Communications Link 4-28

NDS II/Series IV/OpenNET™ Toolbox 4-31

PRODUCT BRIEFS

VAX*/VMS*/OpenNET™ Networking Software 4-36

FACT SHEETS

Networking for the Development Environment 4-39

APPLICATION NOTES

AP-240 Using Archive to Efficiently Control a Network 4-42

AP-242 Additional Printer Support for the NDS-II System 4-54

AP-244 Distributed Job Control the Key to Increased Network Productivity 4-66

AP-246 Setting Up an Efficient Hierarchical File System 4-79

AP-247 Adding Capability to the NDS-11 System with Cluster Boards 4-93

AP-278 Integrating the PC AT into the Intel Development Environment 4-105

ARTICLE REPRINTS

AR-204 Smartlink comes to the Rescue of Software-Development Managers 4-124

AR-425 Helping Computers Communicate 4-129

CHAPTER 5

Microcomputer Development Systems

DATA SHEETS

iMDX 430/431/440/441 Intellec® Series IV Microcomputer Development System 5-1

iPDS™ Personal Development System 5-6

Table of Contents (Continued)

iPDS™-130 Optional Flexible External Disk Drive for the iPDS™ Personal Development System 5-18
iPDS™ Proto Kit 5-22
APPLICATION NOTES
AP-156 Designing Modules for the iPDS™ and iUP Systems 5-24
AP-245 Using Command Files to Speed Program Development 5-46

CHAPTER 6
System Design Kits
DATA SHEETS
SDK-86 MCS®-86 System Design Kit 6-1

CHAPTER 7
PROM Programming
DATA SHEETS
iUP-200A/iUP-201A Universal PROM Programmers 7-1
iUP/iPDS™ Programming Modules 7-11
APPLICATION NOTES
AP-179 PROM Programming with the Intel Personal Development System (iPDS™) 7-20

CHAPTER 8
EPLD Development Tools
DATA SHEETS
iPLDS Intel Programmable Logic Development System 8-1
APPLICATION NOTES
AP-279 Implementing and EPLD Design Using Intel's Programmable Logic Development System 8-9

Alphanumeric Index

386 Software Tools 1-1
80286 Software Development Package 1-14
80287 Support Library 1-77
80386 Development Environment 2-44
8051 Software Packages 1-81
8086 Handyman 2-23
8086/8088/80186/80188 Software Packages 1-40
8086/8088 Software Development Package for Series II/PDS 1-30
8087 Support Library 1-73
Ada 286 Compilation System 1-113
AEDIT Source Code and Text Editor for PC-DOS 2-28
Compilengine iMDX 485CE 4-6
Debug MONitor 386 (D-MON 386P) 2-20
FORTRAN 80 8080/8085 ANS FORTRAN 77 Intellec® Resident Compiler 1-61
I²ICE Integrated Instrumentation and In-Circuit Emulation System 3-9
iC 286 1-26
iMDX 430/431/440/441 Intellec® Series IV Microcomputer Development System 5-1
iMDX 555 NDS-II NRM OpenNET™ Upgrade 4-20
iMDX 581 ISIS Cluster Board Packages 4-24
iPAT™ Performance Analysis Tool 2-31
iPDS™ Personal Development System 5-6
iPDS™ Proto Kit 5-22
iPDS™-130 Optional Flexible External Disk Drive for the iPDS™ Personal Development System 5-18
iPLDS Intel Programmable Logic Development System 8-1
iSBE-96 Development Kit Single Board Emulator 3-1
iUP-200A/iUP-201A Universal PROM Programmers 7-1
iUP/iPDS™ Programming Modules 7-11
Intel Asynchronous Communications Link 4-28
Intel Microprocessor Languages 1-115
ICE™-386 In-Circuit Emulator for the 80386 3-73
ICE™-5100/044 In-Circuit Emulator for the RUPI™-44 Family 3-83
ICE™-5100/252 In-Circuit Emulator for MCS®-51 Family of Microcontrollers 3-91
ICE™-5100/452 In-Circuit Emulator for the UPI™ 452 Family of Programmable I/O Processors 3-101
MCS®-48 Diskette Based Software Package 1-90
MCS®-96 Software Development Packages 1-92
Networking for the Development Environment 4-39
NDS II/Series IV/OpenNET™ Toolbox 4-31
NDS II/VAX* Link Networking Software 4-17
OpenNET™ Network Resource Manager (NRM) iMDX 460 4-1
OpenNET™ Personal Computer Link 4-11
Pascal 80 Software Package 1-65
PASCAL 286 1-19
PL/M 286 1-22
PL/M 80 High-Level Programming Language 1-70
PSCOPE High-Level Program Debugger for iRMX™, XENIX*, Series III and Series IV 2-12
PSCOPE Monitor 386ES (P-MON 386ES) 2-1
PSCOPE-86 for DOS High-Level Application Program Debugger 2-5
SDK-86 MCS®-86 System Design Kit 6-1
VAX*/VMS* Resident Software Development Packages for 80286 1-100
VAX*/VMS* Resident 8086/8088/80186 Software Development Packages 1-106
VAX*/VMS*/OpenNET™ Networking Software 4-36
VLSiCE™-96P In-Circuit Emulator for the MCS®-96 Family of Microcontrollers 3-62

CUSTOMER SUPPORT

CUSTOMER SUPPORT

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, and consulting services. For more information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is quite extensive. It includes factory repair services and worldwide field service offices providing hardware repair services, software support services, customer training classes, and consulting services.

HARDWARE SUPPORT SERVICES

Intel is committed to providing an international service support package through a wide variety of service offerings available from Intel Hardware Support.

SOFTWARE SUPPORT SERVICES

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and COMMENTS Magazine). Basic support includes updates and the subscription service. Contracts are sold in environments which represent product groupings (i.e., iRMX environment).

CONSULTING SERVICES

Intel provides field systems engineering services for any phase of your development or support effort. You can use our systems engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training, and customizing or tailoring an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, bitbus and LAN applications.

INTRODUCTION

Intel recognizes that developing a product based on an advanced microprocessor creates major challenges for an engineering group. Intel helps you meet these challenges and keep your project under control with a set of development tools tailored to the architecture you are using. These tools help you get your product development done with your schedule and budget targets by solving problems that waste valuable engineering time.

The first key to productive development is to work on your product instead of developing and integrating tools. Intel has tools for each phase of your project, and each of them works smoothly with the others to form an effective, integrated tool set. And the tools work on popular industry-standard systems, including the IBM PC AT and PC XT and compatible personal computers and Digital Equipment Corporation VAX/VMS* systems.

HIGH-LEVEL LANGUAGE SUPPORT

Each Intel microprocessor and microcontroller is supported by a set of high-level languages that have the three important elements of well-integrated tools:

The most important integration is between the tools and the processor. Intel assemblers and compilers are optimized around the architectures they support: that means better performance for your product. And efficient compilers mean you can write more of your code in high-level languages instead of assembly language.

Effective coding generally requires a family of compatible translators so that you can draw on the most appropriate language to implement each part of a design. PL/M, Pascal, C, FORTRAN, and assembly language enjoy certain advantages over each other, depending on the application. You can link object modules from any of the Intel translators without further modifications.

The symbolic debugging power of Intel's debuggers is enhanced by communication between the translators and debuggers.

Development Languages and Utilities

Assemblers	All Intel assemblers—and there's one for every major Intel component—provide full macro support.
PL/M	PL/M was the first high-level language designed expressly for microprocessors. It is a procedure-oriented language with data structuring facilities that gives the engineer full control over microprocessor-dependent architecture features. It is one of the most widely used tools in the microprocessor and microcontroller world.
C	C-86 is a true implementation of the C programming language defined by Kernighan and Ritchie. C is known for its flexibility and portability.
Pascal	Pascal-86 and Pascal-286 are supersets of ISO Pascal, with extensions for independent compilation and port I/O. They also embody advanced code optimization techniques to achieve extremely efficient programs.
FORTRAN	FORTRAN-86 and FORTRAN-286 are ANSI-77 standard compilers augmented with full 8087/80287 support and the ability to handle very large arrays (over 64 KB).
Utilities	Intel linkage utilities allow independent assembly and compilation of program modules. Library managers allow the management of standard modules and routines. In the case of the 80286, a system builder is provided to allow easy configuration of a complex, protected, memory-managed system.

HARDWARE AND SOFTWARE DEBUGGERS

Most of the unpleasant surprises that can delay a project attack in the debugging phase. Intel has made debuggers a part of each microprocessor family package, beginning with ICE 80, the world's original in-circuit emulator. Intel's debuggers have the power to let you find bugs early, while they are still cheap and easy to fix, and to find many bugs that would not otherwise be fixed without a major waste of engineering time and schedule time.

Intel's popular ICE™ In-Circuit Emulators continue their key role in development projects, with full-speed, transparent debugging for Intel components. Intel ICE debuggers feature symbolic debugging, the ability to stop execution under user-determined conditions, trace collection, and emulation memory for program execution.

DEBUGGERS FOR 8086, 80186, AND 80286 FAMILY MICROPROCESSOR APPLICATIONS

Intel's debugging product line for the 8086, 80186, and 80286 families of microprocessors features a pair of powerful tools covering the full range of development needs:

Debugging Task	Tool
Host-resident, high-level software debugging	PSCOPE
Full-Speed, transparent software-hardware integration and debug	I²ICE™ Emulator

The tools share a common user interface and high-level language debugging capability. Symbolic debugging automates a task that can eat up valuable development time and introduce error into the debug process. Symbolic debugging builds on the debug records loaded from the output of Intel assemblers and compilers—yet another example of the added debugging power gained from integration of development tools. Using user-defined names, the engineer has access to memory locations and program variables (including dynamic variables and high-level-language data structures).

PSCOPE High-Level Language Debugger

PSCOPE is a host-resident debugger that lets you execute and debug programs at the source code level. You can set break and trace points, examine memory, or simply follow program flow at the instruction, statement or procedure level for programs written in PL/M, Pascal, C, FORTRAN, 8086 assembly language, or 80286 assembly language. PSCOPE even lets you make high-level language patches and store them for later use in updating source files.

The PSCOPE syntax, including debug procedures, is the same used by the I²ICE and TargetSCOPE systems, so that when you move from software development to software-hardware integration, the user interface stays the same. There's no new learning curve to ascend, no lag in the development cycle.

I²ICE™ Integrated Instrumentation and In-Circuit Emulation System

I²ICE is unmatched in its ability to kill hardware and software bugs across the entire development process. Of course, I²ICE offers the high-level language symbolic debugging expected of a software debugger. It also integrates transparent emulation support for all members of Intel's 8086 and 80286 families of microprocessors.

A full I²ICE configuration can simultaneously emulate four separate processors, stopping execution on an individual event, on an address range, on conditional events and on inter-processor events. The system then displays a trace of execution or bus activity. Full-speed execution is possible using either target system memory or up to 288 K-bytes of emulator memory for each processor.

PERFORMANCE ANALYSIS

The iPAT Performance Analysis Tool provides real-time performance analysis and real-time coverage of programs running on 8086/88, 80186/88, and 80286 microprocessors to help software engineers optimize code and improve software reliability.

Object code generated by Intel assemblers and compilers (C, PL.M, Pascal, and FORTRAN) can be analyzed symbolically to improve software efficiency and to validate test coverage. Any object code that lacks compiler information—but that can be run by Intel emulators and for which an absolute program map is available—can also be analyzed non-symbolically by the iPAT analyst.

DEBUGGERS FOR 80386 FAMILY MICROPROCESSOR APPLICATIONS

Users of Intel's 80386 advanced, 32-bit microprocessor have a compatible set of software and hardware debugging tools available for their projects:

Debugging Task	Tool
High-level software debugging	PSCOPE Monitor 386 (P-MON 386)
Software debugging monitor	Debug Monitor 386 (D-MON 386)
Full-speed, transparent software-hardware integration and debug	ICE™ 386 Emulator

PSCOPE Monitor (P-MON 386)

P-MON 386 is a high-level, hosted software debugger for 80386-based systems. It can access and control all of the 80386's visible user hardware resources without any assistance from the operating system. It can also be used to debug applications running under the control of an operating system.

P-MON 386 allows symbolic debugging of programs written in high-level languages. With the help of this debugger, a user can download an application program into the target prototype memory, set hardware and software breakpoints at symbolically specified addresses, trace program execution, and write patches to the program under development.

Debug Monitor 386 (D-MON 386)

D-MON 386 is an unhosted, EPROM-based software debug monitor that provides system-level debug support for 80386 systems. Using D-MON 386, a user can set hardware and software breakpoints, examine and modify memory and registers, and control program execution. This monitor can be configured to run on any 80386-based target board with a user-supplied communication driver and hardware initialization routine.

ICE 386™ In-Circuit Emulator

The ICE™ 386 In-Circuit Emulator provides hardware and software debugging for 80386-based designs. Its capabilities include emulation for the 80386 CPU and the 80287 and 80387 numeric processors. With ICE 386, programs can execute continuously at speeds up to 16 MHz or in a single-step mode. And it includes symbolic debugging to let users work in the context of their original programs.

Intel designed the 80386 and ICE 386 interactively to get the debugging power required of an advanced, 32-bit microprocessor, including non-intrusive access to internal processor activity. Breakpoints allow stopping emulation on specified instruction execution addresses or data addresses. Trace capability lets a user record program execution history prior to the break.

DEBUGGERS FOR MICROCONTROLLER APPLICATIONS

Microcontroller applications are typically characterized by high performance requirements, a variety of asynchronous events, and a lot of on-chip activity. All of these characteristics add to the challenge of debugging your product. Each Intel microcontroller family has in-circuit debugging support to meet the challenges. The ICE and VLSiCE emulators share a user interface with I2ICE and PSCOPE, which saves learning time for projects with multiple processor types.

ICE™ 5100 In-Circuit Emulator

The ICE 5100 emulator gives its user, real-time, non-obtrusive control over 8051-family system debugging at clock speeds up to 16 MHz. It includes the ability to view and modify system activity at a symbolic, high-level language level. ICE 5100/252 debugs HMOS and CHMOS versions of the 8051, the 8052, and the 80C52 including on-chip RAM and ROM. The ICE 5100/044 supports the 8044, including BITBUS™ systems.

VLSiCE™-96 In-Circuit Emulator

VLSiCE 96 provides real-time, non-obtrusive debugging support for the MCS-96 family of 16-bit microcontroller components. It features full symbolic debugging; 64 K-bytes of mappable ICE memory; dynamic execution and data trace, including internal RAM accesses; and a break/state machine which allows stopping emulation or enabling trace on user specified combinations of execution addresses, opcodes, data addresses and values, and selected PSW bits.

iSBE 96 8096 Emulator

The iSBE 96 debugger permits basic execution and debug of programs written for the MCS 96 family of 16-bit microcontrollers, within the emulator or in the user's target system.

GENERAL TOOLS FOR ALL COMPONENT FAMILIES

EPROM Programming Support

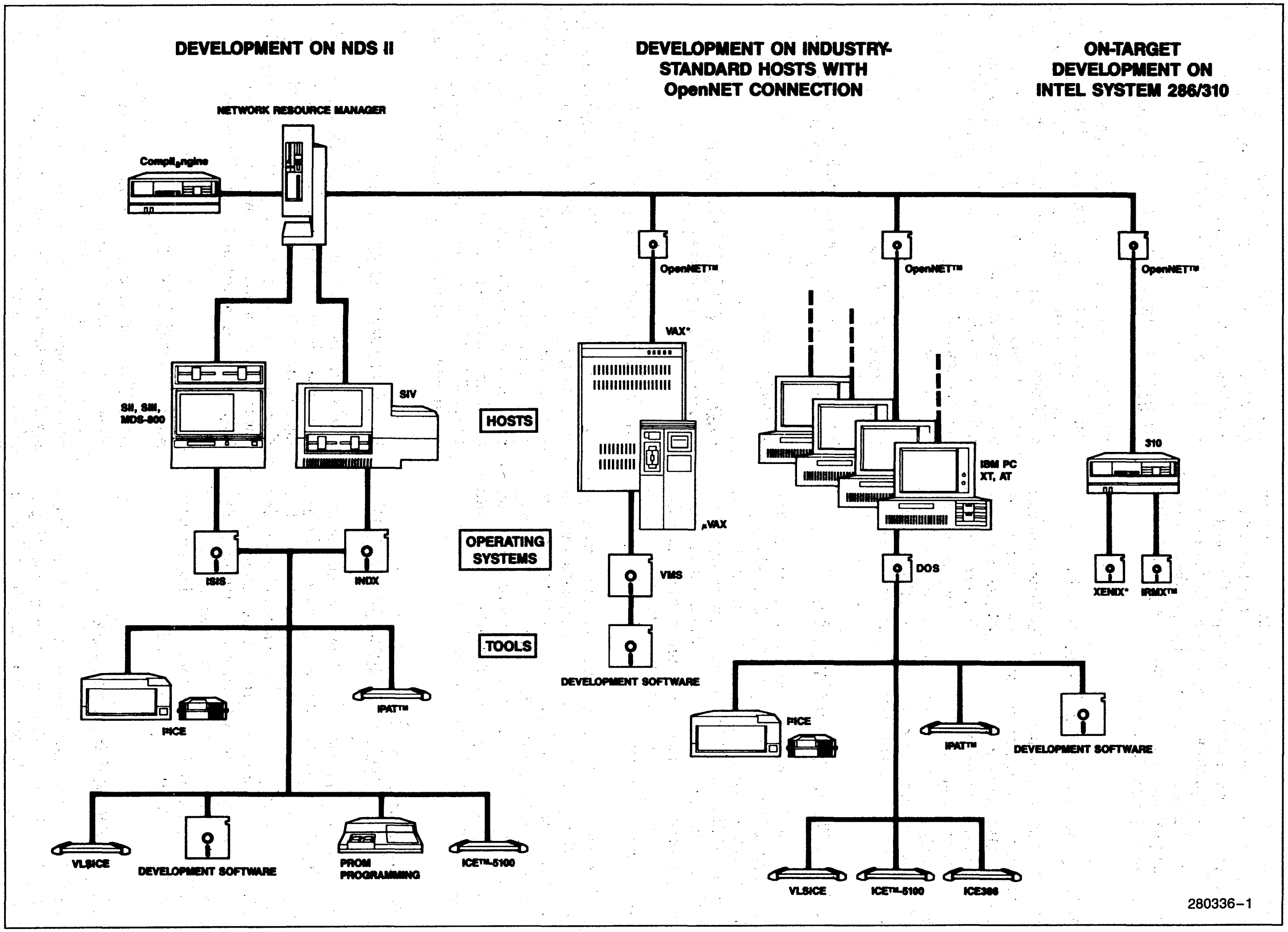
Intel offers a full line of EPROM programmers for Intel devices. Through parallel development efforts, Intel is able to provide the earliest programming support for new Intel EPROMs, EEPROMs, KEPROMs and microcontrollers—with the fastest programming algorithms in the industry. The modular architecture of Intel EPROM programmers allows new support to be added with low-cost add-ons, as they become available.

EPLD Development Tools

Intel's iPLDS Programmable Logic Development System makes it easy to use an erasable, programmable logic device (EPLD) in your design. The iPLDS provides all of the software, programming hardware, and documentation needed to convert random logic into a fully optimized, tested, and document device.

AEDIT Text Editor

AEDIT is a full-screen text editor that can be either menu- or command-driven. It offers the ability to switch easily between two files or to view two files simultaneously through windows. Text entry and editing are further simplified through the use of macros, which allow you to save command clusters for later use.



Tools Support on OpenNET Local Area Network

Development Host Selection

Intel's development tools are available on a selection of industry-standard host systems, giving users of Intel microprocessors and microcontrollers the ability to apply a combination of valuable elements in their development projects:

- design and debug tools built around the needs of the specific microprocessor or microcontroller.
- host systems optimized around installed equipment or the experience and needs of the development team:

VAX/VMS	Centralized development and project control for large teams, on an industry standard system.
PC AT, PC XT (DOS)	Versatile, standard, high-performance workstation.

- continued use of Intellec Series, II, III, and IV and Model 800 dedicated development systems.
- an open network to link tools across the various host environments.

Whether you run the Intel tools on a VAX minicomputer, a PC AT or XT, or an Intel system, the integration work is done before you install the tools on the system—you don't waste time getting the tools ready for the project.

Network Connections

Your host workstations can be a part of a complete development network using Intel's OpenNET™ implementation of the high-performance Ethernet local area network.

The OpenNET network is based on open, ISO OSI standard protocols. In a development application it lets your PC- and VAX-based development stations share files resident on the VAX system. The OpenNET connection also (1) lets PC users share files resident on Intel's NDS II Network Resource Manager and (2) gives users doing on-target development on Intel iRMX® and XENIX* systems access to files resident on a VAX/VMS, iRMX, XENIX, or DOS system from an iRMX, XENIX, or DOS system.

Component Support on Industry Standard Host Systems

	Components Supported				
Development Languages	**8086/80186 8088/80188**	**80286**	**80386**	**8096**	**8044 8051**
Assembler	PC VAX/VMS Series IV	PC VAX/VMS Series IV	PC VAX/VMS	PC Series IV	PC Series II Series IV
PL/M	PC VAX/VMS Series IV	PC VAX/VMS Series IV	PC VAX/VMS	PC Series IV	PC Series II Series IV
C	PC VAX/VMS Series IV	PC VAX/VMS Series IV	PC VAX/VMS	PC	
Pascal	PC VAX/VMS Series IV	PC VAX/VMS Series IV			
FORTRAN	PC VAX/VMS Series IV				
Debuggers					
PSCOPE	PC Series IV				
I2ICE™	PC Series IV	PC Series IV			
VLSiCE™				PC Series IV	
ICE™			PC		PC Series IV

NOTES:
Tools that run on Series IV or Series II also run on Series III.
Intel also offers versions of development languages that run on iRMX™- and Xenix-based systems for on-target development.

*VAX and VMS are trademarks of Digital Equipment Corporation.
*XENIX is a trademark of Microsoft Corporation.

Microcomputer Development Languages

1

386 SOFTWARE TOOLS

PL/M 386 Software Package

- Systems Programming Language for the Protected Virtual Address Mode 386
- Upward Compatible with PL/M 286, PL/M 86, and PL/M 80 Assuring Software Portability

386 Relocation, Linkage and Library Tools

- Provides System Development Capability for High-Performance 386 Applications
- Allows Creation of Multi-User Virtual Memory, and Memory-Protected Systems

C 386

- Implements Full C Language and New Extensions
- Produces High Density Code Rivaling Assembler
- Supports Intel Object Module Format (OMF)

ASM 386

- Instruction Set and Assembler Mnemonics Are Upward Compatible with ASM 286 and ASM 86
- Type-Checking at Assembly Time Helps Reduce Errors at Run-Time

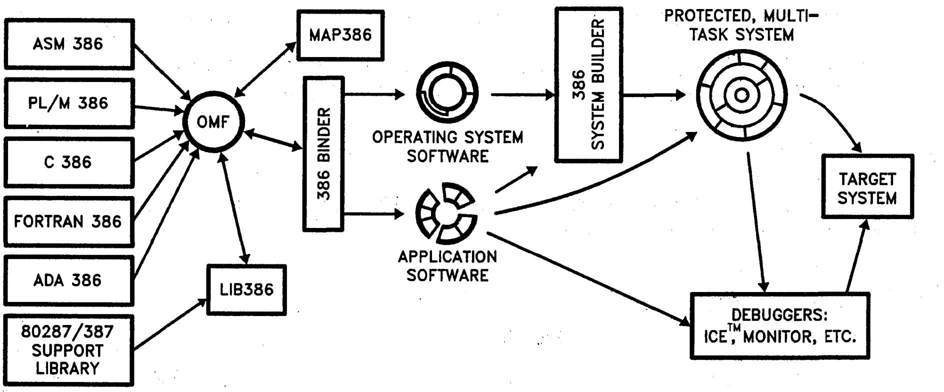


Figure 1. Development Environment Tools for the 386

386 Software tools are available on industry standard hosts, including VAX/VMS, PC-DOS, and XENIX*

November 1986
Order Number: 231637-002

ASM 386

- **Instruction Set and Assembler Mnemonics Are Upward Compatible with ASM 286 and ASM 86**
- **Powerful and Flexible Text Macro Facility**
- **Type-Checking at Assembly Time Helps Reduce Errors at Run-Time**
- **Structures and Records Provide Powerful Data Representation**
- **"High-Level" Assembler Mnemonics Simplify the Language**
- **Supports Full Instruction Set of the 386, Including Memory Protection and Numerics**
- **Supports 286 Addressing Modes**

ASM 386 is the "high-level" macro assembler for the 386 assembly language. ASM 386 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM 286/86/88 mnemonics; new ones have also been added to support the new 386 instructions. The segmentation directives have been greatly simplified.

The 386 assembly language includes approximately 275 instruction mnemonics. From these few mnemonics the assembler can generate over 40,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 275 mnemonics to generate all possible machine instructions. ASM 386 will generate the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM 386 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM 386 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM 386 object modules conform to a thorough, well-defined format used by all 386 high-level languages and utilities. This means it is easy to call (and be called from) HLL object modules.

SUPPORT

Hotline Telephone Support, Software Performance Report (SPR), Software Update, Technical Reports, and Monthly Technical Newsletters are available.

ORDERING INFORMATION

Part Number	Description	Operating Environment
X286ASM386	386 Assembler	286/310 XENIX* System
D86ASM386	386 Assembler	PC-DOS 3.0 or greater

Documentation Package

ASM 386 Assembly Language Reference Manual
ASM 386 Macro Assembler Operating Instructions for XENIX* 286 Systems
ASM 386 Pocket Reference for XENIX 286 Systems

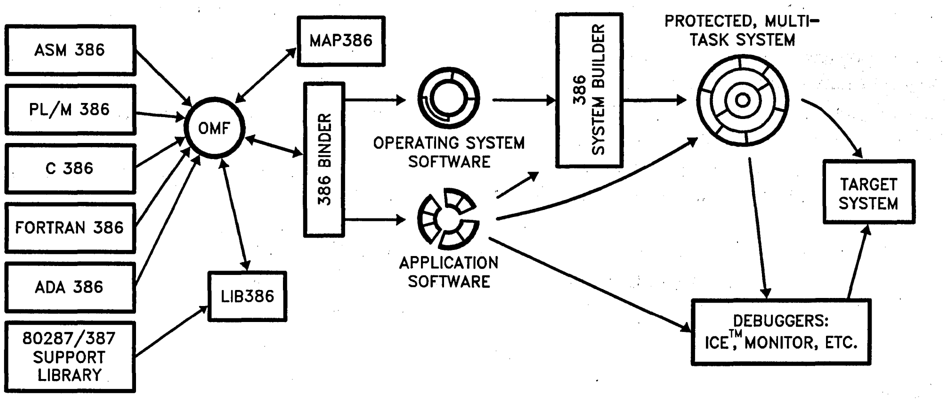
*XENIX™ is a trademark of Microsoft.

386 RELOCATION, LINKAGE AND LIBRARY TOOLS

- **System Development Capability for High-Performance 386 Applications**
- **Allows creation of Multi-User, Virtual Memory, and Memory-Protected Systems**
- **System Utilities for Program Linkage and System Building**
- **Package Supports Program Development with ASM 386, PL/M 386, C 386, Ada 386 and FORTRAN 386.**

The 80386 is a 32-bit microprocessor system with 32-bit addressing, integrated memory protection, and instruction pipelining for high performance. The 386 Relocation, Linkage, and Library Tools are a cohesive set of software design aids for programming the 386 microprocessor system. The package enables system programmers to design protected, multi-user and multi-tasking operating system software, and enables application programmers to develop tasks to run on a protected operating system.

The 386 Relocation, Linkage and Library tools include a program binder (for linking separately compiled modules together), a system builder (for configuring protected multiple-task systems), a cross reference mapper, a program librarian, and the 287/387 support library.



261637-2

Figure 1. Development Environment Tools for the 386

386 SYSTEM BUILDER

- **Supports Complete Creation of Protected, Multi-task Systems**
- **Resolves PUBLIC/EXTERNAL Definitions (between protection levels)**
- **Supports Memory Protection by Building System Tables, Initializing Tasks, and Assigning Protection Rights to Segments**
- **Creates a Memory Image of a 386 System for Cold-start Execution**
- **Target System may be Boot-loadable, Programmed into ROM, or loaded from Mass-store.**
- **Generates Print File with Command Listing and System Map**

BLD 386 is the utility that lets system programmers configure multi-tasking, protected systems from an operating system and discrete tasks. The Builder generates a cold-start execution module, suitable for ROM-based or disk-based systems.

The Builder accepts input modules from 386 translators or the 386 Binder. It also accepts a "Build File" containing definitions and initial values for the 386 protection mechanism - descriptor tables, gates, segments, and tasks. BLD 386 generates a Loadable or bootloadable output module, as well as a print file with a detailed map of the memory-protected system.

Using the Builder command Language, system programmers may perform the following functions:

— Assign physical addresses to segments; also set segment access rights and limits.

— Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.

— Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.

— Support Page tables for boot files.

— Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.

— Create Task State Segments and Task Gates for multi-task applications.

— Resolve inter-module and inter-level references, and perform type-checking.

— Automatically select required modules from libraries.

— Configure the memory image into partitions in the address space.

— Selectively generate an object file and various sections of the print file.

386 BINDER

- **Links Separately Compiled Program Modules Into an Executable Task**
- **Makes the 386 Protection Mechanism Invisible to Application Programmers**
- **Works with PL/M 386, C 386, FORTRAN 386 and ASM 386 Object Modules**
- **Performs Incremental Linking with Output of Binder and Builder**
- **Resolves PUBLIC/EXTERNAL Code and Data References, and Performs Intermodule Type-Checking**
- **Provides Print File Showing Segment Map, Errors and Warnings**
- **Assigns Virtual Addresses to Tasks in the 2^{32} Address Space**
- **Generates Linkable or Loadable Module for Debugging**

The Binder is the only utility an application programmer needs to develop and debug an individual task. Users of the Binder need not be concerned with the architecture of the target machine, making application program development for the 386 very simple.

BND 386 combines 386 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules written in ASM 386, PL/M 386, C 386 and FORTRAN 386 and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND 386 generates a print file displaying a segment map and error messages.

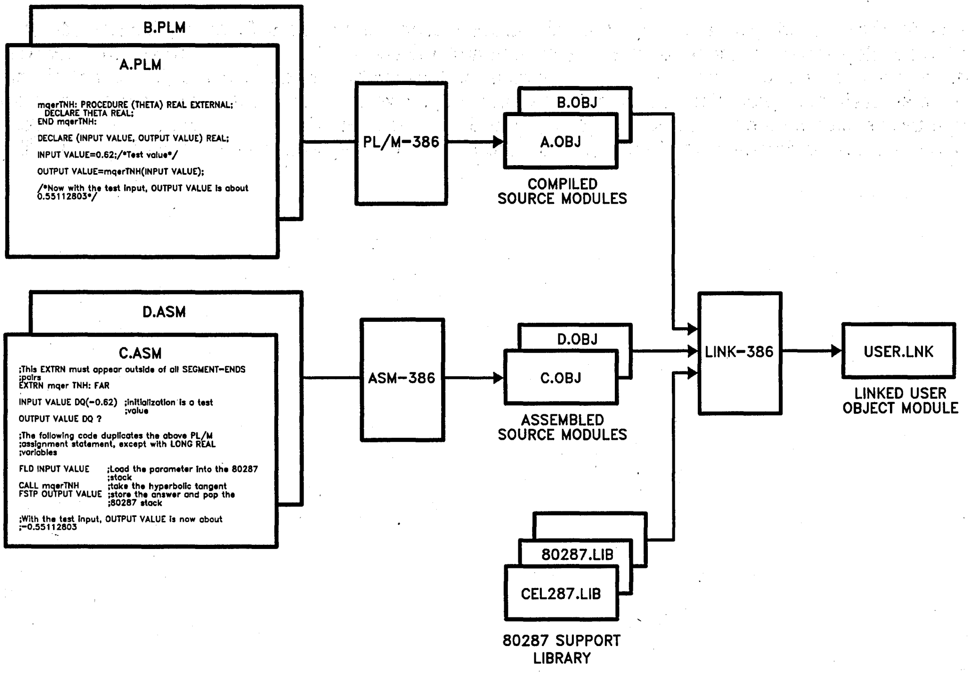
The Binder will be used by system programmers and application programmers. Since application programmers need to develop software independent of any system architecture, the 386 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 386 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of the 386 utilities.

80287 SUPPORT LIBRARY

- **Library to support floating point arithmetic in C 386, PL/M 386, ADA 386, ASM 386, and FORTRAN 386**
- **Decimal conversion module supports binary-decimal conversions**
- **Supports proposed IEEE Floating Point Standard for high accuracy and software portability**
- **Common elementary function library provides trigonometric, logarithmic and other useful functions**
- **Error-handler module simplifies floating point error recovery**

The 80287 Support Library provides C 386, PL/M 386, ADA 386, ASM 386 and FORTRAN 386 users with numeric data processing capability. With the Library, it is easy for programs to do floating point arithmetic. Programs can bind in library modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. Figure 1 below illustrates how the 80287 Support Library can be bound with PL/M 386 and ASM 386 user code to do this. The 80287 Support Library supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable—the software investment is maintained.

The 80287 Support Library consists of the common elementary function library (CEL287.LIB), the decimal conversion library (DC287.LIB), the error handler module (EH287.LIB) and interface libraries (80287.LIB), (NUL287.LIB).



```
mqerTNH: PROCEDURE (THETA) REAL EXTERNAL;
  DECLARE THETA REAL;
END mqerTNH:

DECLARE (INPUT VALUE, OUTPUT VALUE) REAL;

INPUT VALUE=0.62;/*Test value*/

OUTPUT VALUE=mqerTNH(INPUT VALUE);

/*Now with the test input, OUTPUT VALUE is about
0.55112803*/
```

```
;This EXTRN must appear outside of all SEGMENT-ENDS
;pairs
EXTRN mqer TNH: FAR

INPUT VALUE DQ(-0.62)  ;initialization is a test
                       ;value
OUTPUT VALUE DQ ?

;The following code duplicates the above PL/M
;assignment statement, except with LONG REAL
;variables

FLD INPUT VALUE      ;Load the parameter into the 80287
                     ;stack
CALL mqerTNH         ;take the hyperbolic tangent
FSTP OUTPUT VALUE    ;store the answer and pop the
                     ;80287 stack

;With the test input, OUTPUT VALUE is now about
;-0.55112803
```

231637-3

Figure 2. Use of 80287 Support Library with PL/M 386 and ASM 386.

386 MAPPER

- **Flexible Utility to Display Object File Information**
- **MAP 386 Selectively Purges Symbols from a Load Module**
- **Provides Inter-Module Cross-Referencing for Modules Written in All Languages**
- **Supports OS Information**
- **Mapper Allows Users to Display:**

Protection Information	**Debug Information**
SEGMENT TABLES	**MODULE NAMES**
GATE TABLES	**PROGRAM SYMBOLS**
PUBLIC ADDRESSES	**LINE NUMBERS**

The cross-reference map shows references between modules, simplifying debugging. The map also lists and controls all symbolic information in one easy-to-read place.

386 LIBRARIAN

- **Fast, Easy Management of 386 Object Module Libraries**
- **Only Required Modules Are Linked, When Using the Binder or Builder**
- **Librarian Allows Users to: Create Libraries, Add Modules, Replace Modules, Delete Modules, Copy Modules from Another Library, Save Library Module to Object File, Create Backup, Display Module Information (creation date, publics, segments)**

Program libraries improve management of program modules and reduce software administrative overhead. (386 Librarian provides efficient use of program libraries.)

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

ORDERING INFORMATION:

Part Number	Description	Operating Environment
X286RLL386	386 Relocation, Linkage and Library Tools	286/310 XENIX* System
VVSRLL386	386 Relocation, Linkage, and Library Tools	VAX/VMS 4.3 and Later
D86RLL386	386 Relocation, Linkage, and Library Tools	PC-DOS 3.0 or Greater

Documentation Package

386 Utilities User's Guide for Xenix* 286 System
386 System Builder User's Guide for Xenix* 286 System
80287 Support Library Reference Manual

*XENIX is a trademark of Microsoft.

PL/M 386 SOFTWARE PACKAGE

- Systems programming language for the protected virtual address mode 386
- Upward compatible with PL/M 286, PL/M 86 assuring software portability
- Enchanced to support design of protected, multi-user, multi-tasking, virtual memory operating system software
- Produces relocatable object code which is linkable to object modules generated by all other 386 language translators
- Advanced, structured system implementation language for algorithm development
- Supports Intel Object Module Format (OMF)

PL/M 386 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode 386. PL/M 386 has been enhanced to utilize 386 features—memory management and protection—for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M 386 is upward compatible with PL/M 286, PL/M 86 and PL/M 80. Existing systems software can be re-compiled with PL/M 386 to execute in protected virtual address mode on the 80386.

PL/M 386 is the high-level alternative to assembly language programming on the 80386. For the majority of 386 system programs, PL/M 386 provides the features needed to access and to control efficiently the underlying 386 hardware and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M 386 compiler has been designed to efficiently support all phases of software development. Features such as a built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed.

FEATURES

Major features of the Intel PL/M 386 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, for example).

The use of modules and procedures to break down a large problems leads to productive software development. The PL/M 386 implementation of block structure allows the use of REENTRANT procedures, which are especially useful in system design.

Language Compatibility

PL/M 386 object modules are compatible with object modules generated by all other 386 translators. This means that PL/M programs may be linked to programs written in any other 386 languages.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with full symbolic debugging capabilities.

PL/M 386 language is upward compatible with PL/M 286, PL/M 86 and PL/M 80 so that application programs may be easily ported to run on the protected mode 80386.

Supports Fourteen Data Types

PL/M makes use of fourteen data types for various applications. These data types range from one to eight bytes and facilitate various arithmetic, logic, and addressing functions:

—BIT(n):	1 to 32 bit unsigned number
—BYTE:	8 bits unsigned number
—HWORD:	16 bits unsigned number
—WORD:	32 bits unsigned number
—DWORD:	64 bits unsigned number
—OFFSET:	32 bits memory address
—CHARINT:	8 bits signed number
—SHORTINT:	16 bits signed number
—INTEGER:	32 bits signed number
—LONGINT:	64 bits signed number
—REAL:	32 bits floating-point number
—SELECTOR:	16 bits segment name
—POINTER:	48 bits selector, offset
—LONGREAL:	64 bits floating-point number

Another powerful facility allows the use of BASED variables which permit run-time mapping of variables to memory locations. This is especially useful for passing parameters, relative and absolute addressing, and dynamic memory allocation.

Data Type Compatibility

PL/M 286 programs may be recompiled and retargetted to the 386 by use of the WORD16 control. With this control, PL/M 386 provides transparent access to the seven data types provided by PL/M 286.

Two Data Structuring Facilities

In addition to the 14 data types and based variables, PL/M supports two powerful data structuring facilities. These help the user organize data into logical groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of both: Arrays of structures or structures of arrays and structures within structures.

Numerics Support

PL/M programs that use 32-bit REAL data are executed using the 80287 Numeric Data Processor for high performance. All floating-point operations supported by PL/M are executed on the 80287 according to the IEEE floating-point standard. PL/M 386 programs can use built-in functions and predefined procedures—INIT$REAL$MATH$UNIT, SET$REAL$MODE, GET$REAL$ERROR, SAVE$REAL$STATUS, RESTORE$REAL$STATUS—to control the operation of the 80287 within the scope of the language.

Built-In Port I/O

PL/M 386 directly supports input and output from the 386 ports for single BYTE, HWORD and WORD transfers. For BLOCK transfers, PL/M 386 programs can make calls to predefined procedures.

Interrupt Handling

PL/M 386 has the facility for generating and handling interrupts on the 386. A procedure may be defined as an interrupt handler through use of the INTERRUPT attribute. The compiler will then generate code to save and restore the processor status on each execution of the user-defined interrupt handler routine. The PL/M statement CAUSE$INTERRUPT allows the user to trigger a software interrupt from within the program.

Protection Model

PL/M 386 support the implementation of protected operating system software by providing built-in procedures and variables to access the protection mechanism of the 386. Predefined variables—TASK$REGISTER, LOCAL$TABLE, MACHINE$STATUS, CONTROL$REGISTER, etc.—allow direct access and modification of the protection system. Untyped procedures and functions—SAVE$GLOBAL$TABLE, RESTORE$GLOBAL$TABLE, SAVE$INTERRUPT$TABLE, RESTORE$INTERRUPT$TABLE, CLEAR$TASK$SWITCHED$FLAG, GET$ACCESS$RIGHTS, GET$SEGMENT$LIMIT, SEGMENT$READABLE, SEGMENT$WRITABLE, ADJUST$RPL—provide all the facilities needed to implement efficient operating system software.

Compiler Controls

The PL/M 386 compiler offers controls that facilitate such features as:

- Interface to other 386 languages
- Optimization
- Conditional compilation
- The inclusion of additional PL/M source files from disk
- Cross-reference of symbols
- Optional assembly language code in the listing file
- The setting of overflow conditions for run-time handling.
- WORD16/WORD32
- Interface to 286 languages

Addressing Control

The PL/M 386 compiler uses the SMALL and COMPACT controls to generate optimum addressing instructions for programs. Programs of any size can be easily modularized into "subsystems" to exploit the most efficient memory addressing schemes. This lowers total memory requirements and improves run-time execution of programs.

Code Optimization

The PL/M 386 compiler offers four levels of optimization for significantly reducing overall program size.

- Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions
- "Strength reductions": a shift left rather than multiply by 2; and elimination of common subexpressions within the same block
- Machine code optimizations; elimination of superfluous branches; removal of unreachable code
- Optimal local register allocation

Error Checking

The PL/M 386 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed and helpful set of programming and compilation error messages is provided by the compiler and user's guide.

Cost-Effective Alternative to Assembly Language

PL/M 386 programs are code efficient. PL/M 386 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the 386 architecture. Consequently, for the development of systems software, PL/M 386 is the cost-effective alternative to assembly language programming.

Support

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

ORDERING INFORMATION

Part Number	Description	Operating Environment
X286PLM386	PL/M 386 Compiler	XENIX* 286/310
D86PLM386	PL/M 386 Compiler	PC-DOS 3.0 or Greater

Documentation Package
PL/M 386 User's Guide for Xenix* 286 System

*XENIX is a trademark of Microsoft.

C 386
C COMPILER FOR THE 386

- **Implements full C Language**
- **Produces High Density Code Rivaling Assembler**
- **Supports Intel Object Module Format (OMF)**
- **Written in C**
- **Supports IEEE Floating Point Math with 80287 Coprocessor**
- **Supports Bit Fields**
- **Supports Full Standard I/O Library (STDIO)**

Intel C 386 brings the full power of the C programming language to the 386 microprocessor system. Intel C386 supports the full C language as described in the Kernighan and Ritchie book, "The C Programming Language", (Prentice-Hall, 1978). Also included are the latest enhancements to the C language: structure assignments, functions taking structure arguments and returning structures, and the "void" and "enum" data types.

Intel C 386 Compiler Description

The C 386 compiler operates in several phases: preprocessor, parser and code generator. The preprocessor phase interprets directives in C source code, including conditional compilations (# define). The parser phase converts the C program into an intermediate free form and does all syntactic and semantic error checking. The code generator phase converts the parser's output into an efficient intermediate binary code, performs constant folding, and features an extremely efficient register allocator, ensuring high quality code. The code generator outputs relocatable Intel Object Module Format (OMF) code, without creating an intermediate assembly file. The C386 compiler eliminates common code, eliminates redundant loads and stores, and resolves span dependencies (shortens branches) within a program.

The C 386 runtime library consists of a number of functions which the C programmer can call. The runtime system includes the standard I/O library (STDIO), conversion routines, routines for manipulating strings, and (where appropriate) routines for interfacing with the operating system.

C 386 uses Intel's Binder and Builder and generates debug records for symbols and lines on request, permitting access to Intel's PSCOPE Monitor/ICE™ emulator to aid in program testing.

FEATURES

Preprocessor Directives

#define—defines a macro

#include—includes code outside of the program source file

#if—conditionally includes or excludes code

Other preprocessor directives include #undef, #ifdef, #ifdef, #else, #endif, and #line.

Statements

The C language supports a variety of statements:

Conditionals: If, IF-ELSE

Loops: WHILE, DO-WHILE, FOR

Selection of cases: SWITCH, CASE, DEFAULT

Exit from a function: RETURN

Loop Control: CONTINUE, BREAK

Branching: GOTO

Expressions and Operators

The C language includes a rich set of expressions and operators.

Primary expression: invoke functions, select elements from arrays, and extract fields from structures or unions.

Arithmetic operators: add, subtract, multiply, divide, modulus

Relational operators: greater than, greater than or equal, less than, less than or equal, not equal

Unary operators: indirect through a pointer, compute an address, logical negation, ones complement, provide the size in bytes of an operand.

Logical operators: AND, OR

Bitwise operators: AND, exclusive OR, inclusive OR, bitwise complement

Data Types and Storage Classes

Data in C is described by its type and storage class. The type determines its representation and use, and the storage class determines its lifetime, scope, and storage allocation. The following data types are fully supported by C 386.

char: an 8 bit signed integer

int: a 32 bit signed integer

short: a 16 bit signed integer

long: a 32 bit signed integer

unsigned: a modifier for integer data types (char, int, short, and long) which doubles the positive range of values

float: a 32 bit floating point number which utilizes the 80287

double: a 64 bit floating point number

void: a special type that cannot be used as an operand in expressions; normally used for functions called only for effect (to prevent their use in contexts where a value is required).

enum: an enumerated data type

These fundamental data types may be used to create other data types including: arrays, functions, structures, pointers, and unions.

The storage classes available in C 386 include:

register: suggests that a variable be kept in a machine register, often enhancing code density and speed

extern: a variable defined outside of the function where it is declared; retaining its value throughout the entire program and accessible to other modules

auto: a local variable, created when a block of code is entered and discarded when the block is exited

static: a local variable that retains its value until the termination of the entire program

typedef: defines a new data type name from existing data types

BENEFITS

Faster Compilation

Intel C 386 compiles C programs substantially faster than standard C compilers because it produces Intel OMF code directly, eliminating the traditional intermediate process of generating an assembly file.

Portability of Code

Because Intel C 386 supports the STDIO and produces Intel OMF code, programs developed on a variety of machines can easily be transported to the 386.

Full Manipulation of the 386

Intel C 386 enables the programmer to utilize features of the C language to control bit fields, pointers, addresses and register allocation, taking full advantage of the fundamental concepts of the 386.

Support

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

ORDERING INFORMATION

Part Number	Description	Operating Environment
X286C386PP	C 386 Compiler	XENIX* 286/310 System
VVS386	C 386 Compiler	VAX/VMS 4.3 and later
D86C386	C 386 Compiler	PC-DOS 3.0 or greater

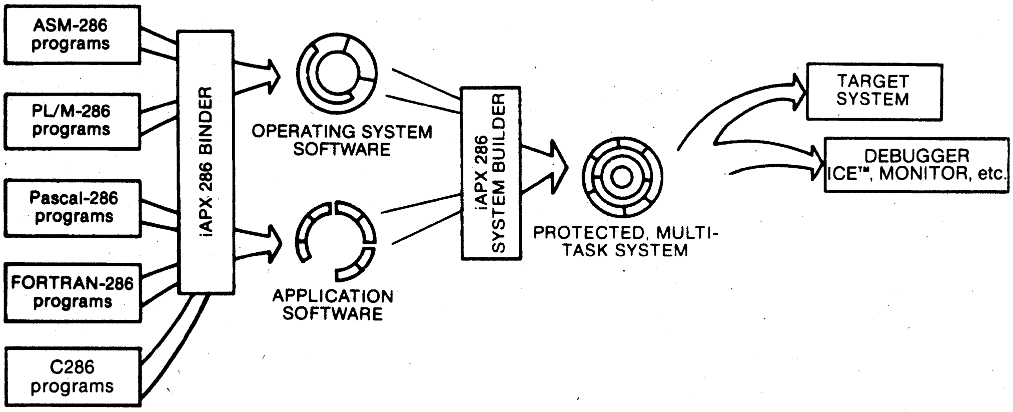
Documentation Package

C 386 User's Guide for Xenix* 286 System

*XENIX is a trademark of Microsoft.

286 SOFTWARE DEVELOPMENT TOOLS AVAILABLE ON CHOICE OF INDUSTRY STANDARD HOSTS INCLUDING PC-DOS AND VAX/VMS*

- **286 Software Development Package**
  - **—Complete System Development Capability for High-Performance 286 Applications**
  - **—Allows Creation of Multi-User, Virtual Memory, and Memory-Protected Systems**
  - **—Macro Assembler for Machine-Level Programming**
- **Pascal-286 Software Package**
  - **—High-Level Programming Language for the Protected Virtual Mode of the 286**
  - **—Implements ISO Standard Pascal**
- **PL/M 286 Software Package**
  - **—Systems Programming Language for the Protected Virtual Address Mode of the 286**
  - **—Advanced Structured System Implementation Language for Algorithm Development**
- **IC-286, C Compiler for the 80286**
  - **—Implements Full C Language**
  - **—Runs Under the Intel UDI, IBM PCs, VAX/VMS*, and Intel Development Systems**



231665-1

The iAPX 286 Software Development Package keeps the protection mechanism invisible to the application programmer, yet easy to configure for the system programmer.

*VAX/VMS are trademarks of Digital Equipment Corporation.

October 1986
Order Number: 231665-002

80286 MACRO ASSEMBLER

- **Instruction Set and Assembler Mnemonics Are Upward Compatible with ASM-86/88**
- **Powerful and Flexible Text Macro Facility**
- **Type-Checking at Assembly Time Helps Reduce Errors at Run-Time**
- **Structures and RECORDS Provide Powerful Data Representation**
- **"High-Level" Assembler Mnemonics Simplify the Language**
- **Supports Full Instruction Set of the 80286/20, Including Memory Protection and Numerics**

ASM-286 is the "high-level" macro assembler for the 80286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new 80286 instructions. The segmentation directives have been greatly simplified.

The 80286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 will generate the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This allows many programming errors to be detected when the program is asembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by all 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

Key Benefit

For programmers who wish to use assembly language. ASM-286 provides many powerful "high-level" capabilities that simplify program development and maintenance.

80286 BINDER

- **Links Separately Compiled Program Modules Into an Executable Task**
- **Makes the 80286 Protection Mechanism Invisible to Application Programmers**
- **Works with PL/M-286, Pascal-286, FORTRAN-286, ASM-286 Object Modules and IC-286**
- **Performs Incremental Linking with Output of Binder and Builder**
- **Resolves PUBLIC/EXTERNAL Code and Data References, and Performs Intermodule Type-Checking**
- **Provides Print File Showing Segment Map, Errors and Warnings**
- **Assigns Virtual Addresses to Tasks in the 2^{32} Address Space**
- **Generates Linkable or Loadable Module for Debugging**

BND-286 is a utility that combines 80286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules written in ASM-286, PL/M-286, Pascal-286, FORTRAN-286 or iC-286 and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder is used by system programmers and application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

Key Benefits

The Binder is the only utility an application programmer needs to develop and debug an individual task. Users of the Binder need not be concerned with the architecture of the target machine, making application program development for the 286 very simple.

80286 MAPPER

- **Flexible Utility to Display Object File Information**
- **MAP-286 Selectively Purges Symbols from a Load Module**
- **Provides Inter-Module Cross-Referencing for Modules Written in All Languages**
- **Mapper Allows Users to Display:**
  Protection Information:
    - **Segment Tables**
    - **Gate Tables**
    - **Public Addresses**

  Debug Information:
    - **Module Names**
    - **Program Symbols**
    - **Line Numbers**

Key Benefit

A cross-reference map showing references *between* modules simplifies debugging; the map also lists and controls all symbolic information in one easy-to-read place.

80286 LIBRARIAN

- **Fast, Easy Management of 80286 Object Module Libraries**
- **Only Required Modules Are Linked, When Using the Binder or Builder**
- **Librarian Allows User to:
  Create Libraries
  Add Modules
  Replace Modules
  Delete Modules
  Copy Modules from Another Library
  Save Library Module to Object File
  Create Backup
  Display Module Information
  (Creation Date, Public, Segments)**

Key Benefit

Program libraries improve management of program modules, and reduce software administrative overhead.

80286 SYSTEM BUILDER

- **Supports Complete Creation of Protected, Multi-Task Systems**
- **Resolves PUBLIC/EXTERNAL Definitions (Between Protection Levels)**
- **Supports Memory Protection by Building System Tables, Initializing Tasks, and Assigning Protection Rights to Segments**
- **Creates a Memory Image of a 286 System for Cold-Start Execution**
- **Target System may be Boot-Loadable, Programmed into ROM, or Loaded From Mass-Store**
- **Generates Print File with Command Listing and System Map**

BLD-286 is the utility that lets system programmers configure mutli-tasking, protected systems from an operating system and discrete tasks. The Builder generates a cold-start execution module, suitable for ROM-based or disk-based systems.

The Builder accepts input modules from 80286 translators or the 80286 Binder. It also accepts a "Build File" containing definitions and initial values for the 286 protection mechanism—descriptor tables, gates, segments, and tasks. BLD-286 generates a Loadable or bootloadable output module, as well as a print file with a detailed map of the memory-protected system.

Using the Builder command Language, system programmers may perfrom the following functions:

— Assign physical addresses to segments; also set segment access rights and limits.
— Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
— Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
— Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
— Create Task State Segments and Task Gates for multi-task applications.
— Resolve inter-module and inter-level references, and perform type-checking.
— Automatically select required modules from libraries.
— Configure the memory image into partitions in the address space.
— Selectively generate an object file and various sections of the print file.

Key Benefit

Allows a system programmer to define the configuration of a protected system in *one* place, with one easy-to-use Utility. This specification may then be adopted by all project members, using either the Builder *or just the Binder*. The flexibility simplifies program development for all users.

SPECIFICATIONS

Documentation

ASM 286 Language Reference Manual
ASM 286 Macro Assembler Operating Instructions
80286 Utilities User's Guide
80286 System Builder User's Guide
Pocket Reference for all the above:
  ASM 286
  Utilties

SUPPORT AVAILABLE

Hotline Telephone Support, Software Updates, Technical Reports

ORDERING INFORMATION

Product Code	Operating Environment
186 ASM 286	Series III/Series IV
D86 ASM 286	IBM PC XT/AT running PCDOS 3.0 or later
iMDX 371 VX	VAX, VMS
X286 ASM 286	Xenix for Intel 286/3XX Systems
R286 ASM 286	RMX 286 for Intel 286/3XX Systems

PASCAL-286 SOFTWARE PACKAGE

- **High-Level Programming Language for the Protected Virtual Mode iAPX 286**
- **Implements ISO Standard Pascal Many Useful Extensions may be Enabled via a Compiler Switch**
- **Choice of Industry Standard Hosts**
- **Supports Full Symbolic Debugging with iAPX 286 Software and ICE™ Debuggers**
- **Upward Compatible with Pascal-86 for Software Portability**
- **Produces Relocatable Object Code Which is Linkable to Object Modules Generated by Other iAPX 286 Translators**
- **Fully Supports the 80287 Numeric Processor using the IEEE Floating Point Standard**

Pascal-286 is a powerful, structured, applications programming language for the protected virtual address mode of the iAPX 286. Pascal-286 is upward compatible with Pascal-86 so that 8086 Pascal source code can be ported to the iAPX 286 in protected mode.

Pascal-286 implements strict ISO standard Pascal, but with many useful extensions. These include separate compilation of modules, interrupt handling, port I/O, and 80287 numerics support. A control is provided in the compiler to flag all non-ISO features used.

Pascal-286 produces relocatable object code which can be linked with object code produced by other iAPX 286 translators such as ASM-286 and PL/M-286. Thus, a combination of translators can be used to provide great programming flexibility.

Type and symbol information needed by software and in-circuit debuggers is added to the object code by the Pascal-286 compiler. This information can be stripped off by the compiler or linker for the final production version.

The Pascal-286 compiler runs on the Intel Microcomputer Development Systems (Series III/Series IV) as well as the IBM PC XT/AT running PCDOS version 3.0 or later.

230863–1

November 1986
Order Number: 230863-001

FEATURES

Conforms to ISO Standard Pascal

Pascal has gained wide acceptance as a portable language for microcomputer applications. However, portability can result only if standards are adhered to. Pascal-286 is a strict implementation of ISO standard Pascal. Extensions are provided to make the language more powerful for microprocessor applications. All extensions are clearly highlighted in the documentation. In addition, the compiler provides a control to flag any non ISO feature used. Pascal-286 will evolve to track future enhancements to standard Pascal.

Upward Compatible with Pascal-86

The Pascal-286 compiler produces object code for the protected virtual address mode of the iAPX 286 language. However, no 286 architecture specific features have been added to the Pascal-286 language. This makes Pascal-286 source code upward compatible with Pascal-86, which allows for porting of 8086 software to the protected 286 with relative ease.

Compatible With Other iAPX 286 Translators

All Intel iAPX 286 translators output object code in a standardized format. This allows 286 programs to be written in a mixture of languages. Systems routines which need access to architectural features can be coded in PL/M-286 or ASM-286. Pascal-286 may be better suited for the applications routines. The systems and application routines can then be combined using the 286 linker (BIND-286).

Standardized Run Time Support

Programs compiled with Pascal-286 can be moved from the development host environment to the target environment with ease. This is the result of standardizing run-time operating system interfaces required by the compiled program into a well defined and well documented set of routines. After programs are developed on a development host, they can then be executed in the target using the same set of system interfaces.

Extensions for Microprocessor Programming

Pascal-286 provides extensions that make it powerful for microprocessor applications. Built-in procedures allow I/O directly from the ports of the iAPX 286. This speeds up I/O as it is done by direct communication with the microprocessor. Interrupt processing is also supported by built in procedures. Examples are: ENABLEINTERRUPTS, DISABLEINTERRUPTS, CAUSEINTERRUPT. Many built in procedures and variables are provided for communicating with the 80287 for numeric computations.

Compiler Controls

The Pascal-286 compiler provides many controls which can be used at invocation time to enhance programming flexibility. Examples are: CODE/NOCODE, DEBUG/NODEBUG, INCLUDE (file), LIST/NOLIST, OPTIMIZE (n), EXTENSIONS/NOEXTENSIONS. All controls have default values that are active unless the opposite is specified during invocation. Thus, for most compiles, no controls need be specified.

Support for IEEE Standard Numerics

Pascal-286 provides full support for the 80287 numerics co-processor. All floating point operations are done according to the IEEE floating point standard. The benefits are predictable, accurate and consistent results. Built-in procedures to support the 80287 include GET8087ERRORS and MASK 8087ERRORS. A full set of 80287 library routines are supplied with the compiler.

Optimizations

The Pascal-286 compiler produces highly optimized code, both in size and execution time. This is achieved by:

— Use of powerful iAPX 286 instructions, in particular, for string handling, 80287 numerics and subroutine linkage
— Short circuit evaluation of boolean expressions, constant folding and strength reduction of multiplications and additions
— Elimination of superfluous branches, optimization of span dependent jumps

Support Available

Hotline service, Software Updates and technical newsletters.

ORDERING INFORMATION

Product Code	Operating Environment	Documentation Package
I86 PAS 286	Intel Series III/Series IV	Pascal-286 User's Guide
D86 PAS 286	IBM PC XT/AT running PCDOS version 3.0 or later	Pascal-286 Pocket Reference

PL/M 286 SOFTWARE PACKAGE

- **System Programming Language for the Protected Virtual Address Mode 80286**
- **Upward Compatible with PL/M 86 and PL/M 80 Assuring Software Portability**
- **Enhanced to Support Design of Protected, Multi-User, Multi-Tasking, Virtual Memory Operating System Software**
- **Multiple Levels of Optimization**
- **Advanced, Structured System Implementation Language for Algorithm Development**
- **Produces Relocatable Object Code Which is Linkable to Object Modules Generated by all Other 80286 Language Translators**
- **Wide Choice of Industry Standard Hosts**

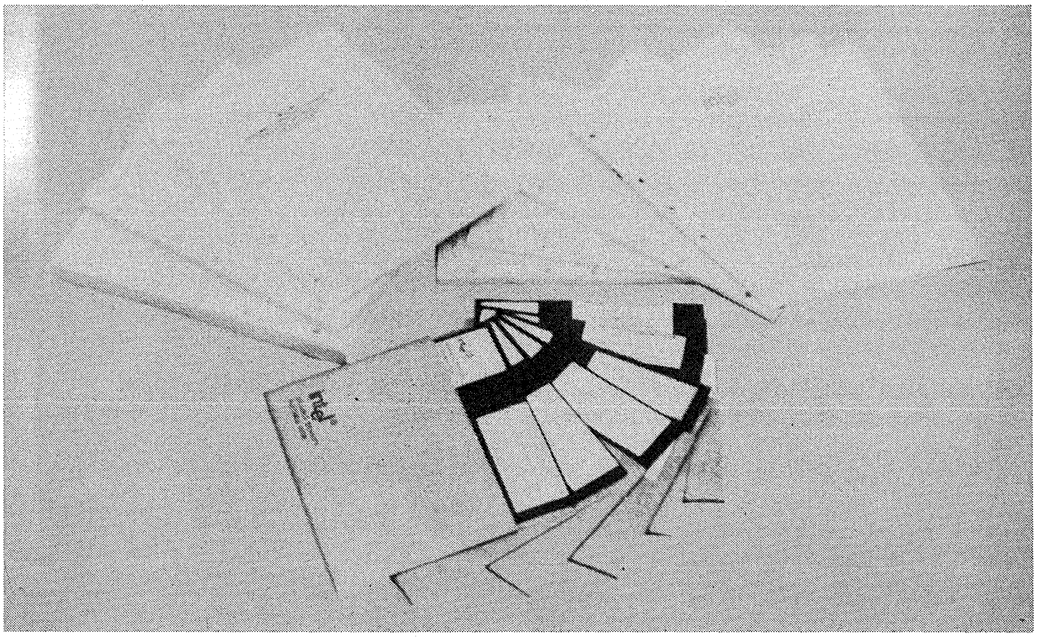
PL/M 286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode 80286. PL/M 286 has been enhanced to utilize 80286 features—memory management and protection—for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M 286 is upward compatible with PL/M 86 and PL/M 80. Existing systems software can be recompiled with PL/M 286 to execute in protected virtual address mode on the 80286.

PL/M 286 is the high-level alternative to assembly language programming on the 80286. For the majority of 80286 systems programs, PL/M 286 provides the features needed to access and to control efficiently the underlying 80286 hardware and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M 286 compiler has been designed to efficiently support all phases of software development features such as a built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed.

The PL/M 286 compiler runs on the Intel Microcomputer Development Systems (Series III/Series IV) as well as the IBM PC XT/AT running PC DOS version 3.0 or later, Digital Equipment VAX/VMS† Systems, and Intel XENIX** 286 and RMX 286 based systems.


280335–01

†VAX, VMS are trademarks of Digital Equipment Corporation.
**XENIX is a trademark of Microsoft Corporation.

December 1986
Order Number: 280335-001

FEATURES

Major features of the Intel PL/M 286 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, for example).

The use of modules and procedures to break down a large problem leads to productive software development. The PL/M 286 implementation of block structure allows the use of REENTRANT procedures, which are especially useful in system design.

Language Compatibility

PL/M 286 object modules are compatible with object modules generated by all other 286 translators. This means that PL/M programs may be linked to programs written in any other 286 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with full symbolic debugging capabilities.

PL/M 286 language is upward compatible with PL/M 86 and PL/M 80 so that application programs may be easily ported to run on the protected mode 80286.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes and facilitate various arithmetic, logic, and addressing functions:

- Byte: 8-bit unsigned number
- Word: 16-bit unsigned number
- Dword: 32-bit unsigned number
- Integer: 16-bit signed number
- Real: 32-bit floating-point number
- Pointer: 16-bit or 32-bit memory address indicator
- Selector: 16-bit pointer base

Another powerful facility allows the use of BASED variable which permit run-time mapping of variables to memory locations. This is especially useful for passing parameters, relative and absolute addressing, and dynamic memory allocation.

Two Data Structuring Facilities

In addition to the seven data types and based variables, PL/M supports two powerful data structuring facilities. These help the user to organize data into logical groups.

- Array: Indexed list of same type data elements
- Structure: Named collection of same or different type data elements
- Combinations of both: Arrays of structures or structures of arrays

Numerics Support

PL/M programs that use 32-bit REAL data are executed using the 80287 Numeric Data Processor for high performance. All floating-point operations supported by PL/M are executed on the 80287 according to the IEEE floating-point standard. PL/M 286 programs can use built-in functions and predefined procedures—INIT$REAL$MATH$UNIT, SET$REAL$MODE, GET$REAL$ERROR, SAVE$REAL$STATUS, RESTORE$REAL$STATUS,—to control the operation of the 80287 within the scope of the language.

Built-In String Handling Facilities

The PL/M 286 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Built-In Port I/O

PL/M 286 directly supports input and output from the 80286 ports for single BYTE and WORD transfers. For BLOCK transfers, PL/M 286 programs can make calls to predefined procedures.

Interrupt Handling

PL/M 286 has the facility for generating and handling interrupts on the 80286. A procedure may be defined as an interrupt handler through use of the INTERRUPT attribute. The compiler will then generate code to save and restore the processor status on each execution of the user-defined interrupt handler routine. The PL/M statement CAUSE$INTERRUPT allows the user to trigger a software interrupt from within the program.

Protection Model

PL/M 286 supports the implementation of protected operating system software by providing built-in procedures and variables to access the protection mechanism of the 80286. Predefined variables—TASK$REGISTER, LOCAL$TABLE, MACHINE$-STATUS, etc.—allow direct access and modification of the protection system. Untyped procedures and functions—SAVE$GLOBAL$TABLE, RESTORE$-GLOBAL$TABLE, SAVE$INTERRUPT$TABLE, RESTORE$INTERRUPT$TABLE, CLEAR$TASK$-SWITCHED$FLAG, GET$ACCESS$RIGHTS, GET$SEGMENT$LIMIT, SEGMENT$READABLE, SEGMENT$WRITEABLE, ADJUST$RPL—provide all the facilities needed to implement efficient operating system software.

Compiler Controls

The PL/M 286 compiler offers controls that facilitate such features as:

— Optimization
— Conditional compilation
— The inclusion of additional PL/M source files from disk
— Cross-reference of symbols
— Optional assembly language code in the listing file
— The setting of overflow conditions for run-time handling

Addressing Control

The PL/M 286 compiler uses the SMALL, COMPACT, MEDIUM, and LARGE control to generate optimum addressing instructions for programs. Programs of any size can be easily modularized into "subsystems" to exploit the most efficient memory addressing schemes. This lowers total memory requirements and improves run-time execution of programs.

Code Optimization

The PL/M 286 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions
— "Strength reductions": a shift left rather than multiply by 2; and elimination of common sub-expressions within the same block
— Machine code optimizations; elimination of superfluous branches; reuse of duplicate code; removal of unreachable code
— Optimization of based-variable operations and cross-statement load/store

Error Checking

The PL/M 286 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed and helpful set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 286 is designed to be an efficient, cost-effective solution to the special requirements of protected mode 80286 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 286 is easy to learn and use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 286, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 286 is designed to aid in the development of reliable software (PL/M 286 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

Cost-Effective Alternative to Assembly Language

PL/M 286 programs are code efficient. PL/M 286 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the 80286 architecture. This includes language features for control of the 80286 protection mechanism. Consequently, for the development of systems software, PL/M 286 is the cost-effective alternative to assembly language programming.

SPECIFICATIONS

Support Available

90 Days:
Hotline Telephone Support, Software Updates, Subscription Service

Documentation Package

PL/M 286 User's Guide
PL/M 286 Pocket Reference

ORDERING INFORMATION

Ordering Code	Operating Environment
I86PLM286	Intel Series III/Series IV
D86PLM286	IBM PC XT/AT running PCDOS version 3.0 or later
iMDX373VX	VAX, VMS
X286PLM286	Xenix for Intel Systems 286/3XX
R286PLM286	iRMX™ 286 for Intel Systems 286/3XX

iC-286
C COMPILER FOR THE 80286

- **Implements Full C Language**
- **Produces High Density Code Rivaling Assembler**
- **Supports Intel Object Module Format (OMF)**
- **Runs under the Intel UDI on Intel Development Systems and iRMX™ 286**
- **Available for the VAX/VMS* Operating System and for PCDOS**
- **Supports Both Small and Large Models of Computation**
- **Supports PSCOPE and I²ICE™**
- **Supports IEEE Floating Point Math with Intel Math Coprocessor**
- **Supports Bit Fields**
- **Supports Full Standard I/O Library (STDIO)**
- **Written in C**

The C Programming Language was originally designed in 1972 and has become increasingly popular as a systems development language. C is not a "very high level" language and is not tied to any specific application area. Although it is used for writing operations systems, it has been used equally well to write numerical, text-processing and data base programs. C combines the flexibility and programming speed of a higher level language with the efficiency and control of assembly language.

Intel iC-286 brings the full power of the C programming language to 80286 based microprocessor systems.

Intel iC-286 supports the full C language as described in the Kernighan and Ritchie book, "The C Programming Language", (Prentice-Hall, 1978). Also included are the latest enhancements to the C language; structure assignments, functions taking structure arguments and returning structures, and the "void" and "enum" data types.

C is rapidly becoming the standard microprocessor system implementation language because it provides:

1. the ability to manipulate the fundamental objects of the machine (including machine addresses) as easily as assembly language.
2. the power and speed of a structured language supporting a large number of data types, storage classes, expressions and statements.
3. processor independence (most programs developed for other processors can be easily transported to the 80286), and
4. code that rivals assembly language in efficiency.

INTEL iC-286 COMPILER DESCRIPTION

The iC-286 compiler operates in four phases; preprocessor, parser, code generator, and optimizer. The preprocessor phase interprets directives in C source code, including conditional compilations (# define). The parser phase converts the C program into an intermediate free form and does all syntactic and semantic error checking. The code generator phase converts the parser's output into an efficient intermediate binary code, performs constant folding, and features an extremely efficient register allocator, ensuring high quality code. The optimizer phase converts the output of the code generator into relocatable Intel Object Module Format (OMF) code, without creating an intermediate assembly file. Optionally, the iC-286 compiler can produce a symbolic assembly like file. The iC-286 optimizer eliminates common code, eliminates redundant loads and stores, and resolves span dependencies (shortens branches) within a program.

The iC-286 runtime library consists of a number of functions which the C programmer can call. The runtime system includes the standard I/O library (STDIO), conversion routines, routines for manipulating strings, special routines to perform functions not available on the 80286 (32-bit arithmetic and emulated floating point), and (where appropriate) routines for interfacing with the operating system.

iC-286 uses Intel's linker and locator and generates debug records for symbols and lines on request, permitting access to Intel's PSCOPE and I²ICE to aid in program testing.

November 1986
Order Number: 280334-001

FEATURES

Support for Small and Large Models

Intel iC-286 supports both the SMALL and LARGE modes of segmentation. A SMALL model program can have up to 64K bytes of code and 64K bytes of data, with all pointers occupying two bytes. Because two byte pointers permit the generation of highly compact and efficient code, this model is recommended for programs that can meet the size restrictions. The LARGE segmentation model is used by programs that require access to the full addressing space of the 80286 processors. In this model, each source file generates a distinct pair of code and data segments of up to 64K bytes in length. All pointers are four bytes long.

Preprocessor Directives

#define—defines a macro
#include—includes code outside of the program source file
#if—conditionally includes or excludes code
Other preprocessor directives include #undef, #ifdef, #ifndef, #else, #endif, and #line.

Statements

The C language Supports a variety of statements:

Conditionals; IF, IF-ELSE
Loops: WHILE, DO-WHILE, FOR
Selection of cases: SWITCH, CASE DEFAULT
Exit from a function: RETURN
Loop control: CONTINUE, BREAK
Branching: GOTO

Expressions and Operators

The C language includes a rich set of expressions and operators.

Primary expression: invoke functions, select elements from arrays, and extract fields from structures or unions

Arithmetic operators: add, subtract, multiply, divide, modulus

Relational operators: greater than, greater than or equal, less than, less than or equal, not equal

Unary operators: indirect through a pointer, compute an address, logical negation, ones complement, provide the size in bytes of an operand.

Logical operators: AND, OR

Bitwise operators: AND, exclusive OR, inclusive OR, bitwise complement

Data Types and Storage Classes

Data in C is described by its type and storage class. The type determines its representation and use, and the storage class determines its lifetime, scope, and storage allocation. The following data types are fully supported by iC-286.

char
an 8-bit signed integer

int
a 16-bit signed integer

short
same as int (on the 80286)

long
a 32-bit integer

unsigned
a modifier for integer data types (char, int, short, and long) which doubles the positive range of values

float
a 32-bit floating point number which utilizes the 80287 or a software floating point library

double
a 64-bit floating point number

void
a special type that cannot be used as an operand in expressions; normally used for functions called only for effect (to prevent their use in contexts where a value is required).

enum
an enumerated data type
These fundamental data types may be used to create other data types including: arrays, functions, structures, pointers, and unions.

The storage classes available in iC-286 include:

register
suggests that a variable be kept in a machine register, often enhancing code density and speed

extern
a variable defined outside of the function where it is declared; retaining its value throughout the entire program and accessible to other modules

auto
a local variable, created when a block of code is entered and discarded when the block is exited

static
a local variable that retains its value until the termination of the entire program

typedef
defines a new data type name from existing data types

BENEFITS

Faster Compilation

Intel iC-286 compiles C programs substantially faster than standard C compilers because it produces Intel OMF code directly, eliminating the traditional intermediate process of generating an assembly file.

Portability of Code

Because Intel iC-286 supports the STDIO and produces Intel OMF code, programs developed on a variety of machines can easily be transported to the 80286.

Rapid Program Development

Intel iC-286 provides the programmer with detailed error messages and access to PSCOPE and I^{2}ICE to speed program development.

Full Manipulation of the 80286

Intel iC-286 enables the programmer to utilize features of the C language to control bit fields, pointers, addresses and register allocation, taking full advantage of the fundamental concepts of the 80286.

SPECIFICATIONS

Operating Environment

The iC-286 compiler runs host resident on both the Intel Series III Microcomputer Development System under ISIS-II and on the System 286/310 under the iRMX™ 286 operating system iC-286 can also run as a cross compiler on a VAX 11/780 computer under the VMS operating system 128K bytes of User Memory is required on all versions. The PCDOS system is also a supported environment. Specify desired version when ordering.

Required Hardware

Development System Version

- Intellec® Microcomputer Development System; Series III or Series IV
- Dual Diskette Drives, Single or Double Density
- System Console; CRT or Hardcopy Interactive Device

iRMX 286 version:

- Any iAPX 286, iSBC® 286 or based system capable of running the iRMX 286 Operating System

VAX version:

- Digital Equipment Corporation VAX 11/780 or compatible computer

PCDOS version:

- PC XT or AT using PCDOS V3.0 or later

Optional Hardware

ISIS-II version:

- ICE-86, I^{2}ICE-86

iRMX-286 version:

- Numeric Data Processors for support of the REALMATH standard

VAX version:

- None

Required Software

ISIS-II version:

- ISIS-II Diskette Operating System
- Series III or Series IV Operating

iRMX 286 version:

- iRMX 286 Realtime Multiprogramming Operation
- iRMX 286 Utilities Package

VAX version:

- VMS Operating System

Optional Software

Development System version:

- None

iRMX 286 version:

- None

VAX version:

- MDS*-384 Kit-Mainframe Link for distributed development, or iMDX-394 Asynchronous Communications Link.
- VAX iAPX 286 MACRO Assembler and utilities package (iMDX-371VX)

Shipping Media

Development System version:

— Two single and one double density ISIS-II format 8″ diskettes, one 5¼″ Series IV Format

iRMX 286 version:

— Double Density iRMX 286 format 5¼″ diskette

VAX version:

— 1600 bpi, 9 track Magnetic tape

DOS version:

— Double Density PC-DOS format 5¼″ diskette

ORDERING INFORMATION

Order Code	Description
i86C286	iC-286 Compiler for ISIS-II, Series IV
R286C286	iC-286 Compiler for iRMX 86
iMDX-377	iC-286 Cross Compiler for VAX/VMS
D86C286	iC-286 Cross Compiler for PCDOS

Intel Software License required.

Documentation Package

The C Programming Language by Kernighan and Ritchie (1978 Prentice-Hall)

iC-286 User Manual

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation. VAX, VMS are trademarks of Digital Equipment Corporation.

intel®

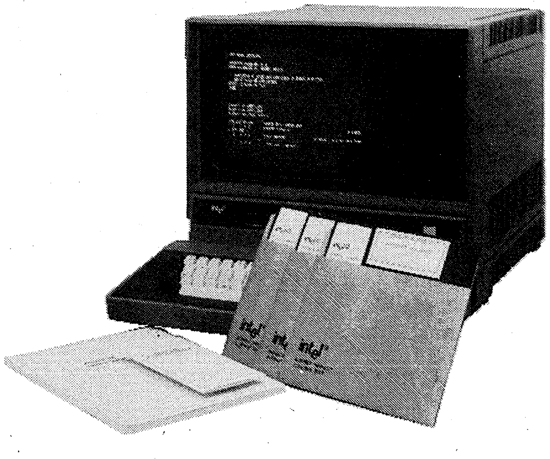
8086, 8088 SOFTWARE DEVELOPMENT PACKAGES FOR SERIES II/PDS

- **PL/M 86/88 High Level Programming Language**
- **ASM 86/88 Macro Assembler for 8086, 8088 Assembly Language Programming**
- **LINK 86/88 and LOC 86/88 Linkage and Relocation Utilities**
- **CONV 86/88 Converter for Conversion of 8080/8085 Assembly Language Source Code to 8086, 8088 Assembly Language Source Code**
- **OH 86/88 Object-to-Hexadecimal Converter**
- **LIB 86/88 Library Manager**

The 8086/8088 Software Development Packages for Series II provide a set of software development tools for the 8086, 8088 CPUs and the iSBC 86/12A single board computer. The packages operate under the ISIS-II operating system on Intel Microcomputer Development Systems—Model 800, Series II or the Personal Development System (PDS)—thus minimizing requirements for additional hardware or training for Intel Microcomputer Development System users.

These packages permit 8080/8085 users to efficiently upgrade existing programs into 8086/8080 code from either 8080/8085 assembly language source code or PL/M 80 source code.

For the new Intel Microcomputer Development System user, the packages operating on a PDS or an Intellec Series II, such as a Model 235, provide total 8086, 8088 software development capability.



280380-1

December 1986
Order Number: 280380-002

PL/M 86/88 COMPILER FOR SERIES II/PDS

- **Language is Upward Compatible from PL/M 80, Assuring MCS-80/85 Design Portability**
- **Supports 16-bit Signed Integer and 32-bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-to-Learn, Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other 8086 Object Modules**
- **Supports Full Extended Addressing Features of the 8086/10 and 8088/10 Microprocessors (Up to 1 Mbyte)**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**

Like its counterpart for MCS-80/85 program development, PL/M 86/88 is an advanced, structured high-level programming language. The PL/M 86/88 compiler was created specifically for performing software development for the Intel 8086, 8088 Microprocessors.

PL/M 86/88 has significant new capabilities over PL/M 80 that take advantage of the new facilities provided by the 8086, 8088 microsystem, yet the PL/M 86/88 language remains compatiable with PL/M 80.

With the exception of hardware-dependent modules, such as interrupt handlers, PL/M 80 applications may be recompiled with PL/M 86/88 with little need for modification. PL/M 86/88, like PL/M 80, is easy to learn, facilitates rapid program development, and reduces program maintenance costs.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or asembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86/88 compiler efficiently converts free-form PL/M language statements into equivalent 86/88 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

FEATURES

Major features of the Intel PL/M 86/88 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible by clearly defining the scope of user variables (local to a private procedure, global to a public module, for example).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86/88 implementation of a block structure allows the use of REENTRANT which is especially useful in system design.

Language Compatibility

PL/M 86/88 object modules are compatible with object modules generated by all other 86/88 translators. This means that PL/M programs may be linked to programs written in any other 86/88 language.

Object modules are compatible with ICE-88 and ICE-86 units; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86/88 Language is upward-compatible with PL/M 80, so that application programs may be easily ported to run on the 8086 or 8080.

Supports Five Data Types

PL/M makes use of five data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

— Byte: 8-bit unsigned number
— Word: 16-bit unsigned number
— Integer: 16-bit signed number
— Real: 32-bit floating point number
— Pointer: 16-bit or 32-bit memory address indicator

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

— Array: Indexed list of same type data elements
— Structure: Named collection of same or different type data elements
— Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the 8087 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or emulator takes place at link-time, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86/88 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for generating interrupts to the 8086 or 8088 via software. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to same and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET$INTERRUPT, the function retuning an INTERRUPT$PTR, and the PL/M statement CAUSE$INTERRUPT all add flexibility to user programs involving interrupt handling.

Segmentation Control

The PL/M 86/88 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64 KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86/88 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions.
— "Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block.
— Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreadable code.
— Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations.

Compiler Controls

The PL/M 86/88 compiler offers more than 25 controls that facilitate such features as:

— Conditional compilation
— Intra- and Inter-module cross reference
— Corresponding assembly language code in the listing file
— Setting overflow conditions for run-time handling

BENEFITS

PL/M 86/88 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 or 88 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 86/88 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86/88, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 86/88 is designed to aid in the development of reliable software (PL/M 86/88 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

8086, 8088 MACRO ASSEMBLER FOR SERIES II/PDS

- **Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **"Strongly Typed" Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as "STRUCTUREs" and "RECORDs"**
- **Over 120 Detailed and Fully Documented Error Messages**
- **Produces Relocatable and Linkable Object Code**

ASM 86/88 is the "high-level" macro assembler for the 8086/88 assembly language. ASM 86/88 translates symbolic 86/10, 88/10 assembly language mnemonics into 86/10, 88/10 relocatable object code.

ASM 86/88 should be used where maximum code efficiency and hardware control is needed. The 8086, 8088 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 86/10, 88/10 machine instructions. ASM 86/88 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characterisitics of forward referenced symbols.

ASM 86/88 offers many features normally found only in high-level languages. The 8086, 8088 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

FEATURES

Major features of the Intel 8086/8088 assembler and assembly language include:

Powerful and Flexible Test Macro Facility

- Macro calls may appear anywhere
- Allows user to define the syntax of each macro
- Built-in functions
  conditional assembly (IF-THEN-ELSE, WHILE)
  repetition (REPEAT)
  string processing functions (MATCH)
  support of assembly time I/O to console (IN, OUT)
- Three Macro Listing Options include a GEN mode which provides a complete trace of all macro calls and expansions.

High-Level Data Structuring Capability

- STRUCTURES: Defined to be a template and then used to allocate storage. The familiar dot notation may be used to form instruction addresses with structure fields.
- ARRAYS: Indexed list of same type data elements.
- RECORDS: Allows bit-templates to be defined and used as instruction operands and/or to allocate storage.

Fully Supports 8086, 8088 Addressing Modes

- Provides for complex address expressions involving base and indexing registers and (structure) field offsets.
- Powerful EQU facility allows complicated expressions to be named and the name can be used as a synonym for the expression throughout the module.

Powerful STRING MANIPULATION INSTRUCTIONS

- Permit direct transfers to or from memory or the accumulator.
- Can be prefixed with a repeat operator for repetitive execution with a count-down and condition test.

Over 120 Detailed Error Messages

- Appear both in regular list file and error print file.
- User documentation fully explains the occurrence of each error and suggests a method to correct it.

Support for ICE-86™ Emulation and Symbolic Debugging

- Debug options for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging.

Generates Relocatable and Linkable Object Code—Fully Compatible with LINK 86/88, LOC 86/88 and LIB 86/88

- Permits ASM 86/88 programs to be developed and debugged in small modules. These modules can be easily linked with other ASM 86/88 or PL/M 86/88 object modules and/or library routines to form a complete application system.

BENEFITS

The 8086/8088 macro assembler allows the extensive capabilities of the 86/88 CPU's to be fully exploited. In any application, time and space critical routines can be effectively written in ASM 86/88. The 86/88 assembler outputs relocatable and linkable object modules. These object modules may be easily combined with object modules written in PL/M 86/88—Intel's structured, high-level programming language. ASM 86/88 compliments PL/M 86/88 as the programmer may choose to write each module in the language most appropriate to the task and then combine the modules into the complete applications program using the 8086/8088 relocation and linkage utilities.

CONV 86/88
MCS®-80/85 TO 8086, 8088 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- **Translates 8080/8085 Assembly Language Source Code to 8086, 8088 Assembly Language Source Code**
- **Provides a Fast and Accurate Means to Convert 8080/8085 Programs to the 8086, 8088 Facilitating Program Portability**
- **Automatically Generates Proper ASM 86/88 Directives to Set Up a "Virtual 8080" Environment that is Compatible with PL/M 86/88**

In support of Intel's commitment to software portability, CONV 86/88 is offered as a tool to move 8080/8085 programs to the 8086, 8088. A comprehensive manual, "MCS-86 Assembly Language Coverter Operating Instructons for ISIS-II Users", covers the entire conversion process. Detailed methodology of the conversion process is fully described therein.

— CONV 86/88 will accept as input an error-free 8080/8085 assembly-language source file and optional controls, and produce as output, optional PRINT and OUTPUT files.

— The PRINT file is a formatted copy of the 8080/8085 source and the 86/88 source file with embedded caution messages.

— The OUTPUT file is an 86/88 source file.

— CONV 86/88 issues a caution message when it detects a potential problem in the converted 86/88 code.

— A transliteration of the 8080/8085 programs occurs, with each 8080/8085 construct mapped to its exact 86/88 counterpart:

Registers
Condition flags
Instruction
Operands
Assembler directives
Assembler control lines
Macros

Because CONV 86/88 is a transliteration process, there is the possibility of as much as a 15%-20% code expansion over the 8080/8085 code. For compactness and efficiency it is recommended that critical portions of programs be re-coded in 8086, 8088 assembly language.

Also, as a consequence of the transliteration, some manual editing may be required for converting instruction sequences dependent on:

— instruction length, timing, or encoding

— interrupt processing*

— PL/M parameter passing conventions*

*Mechanical editing procedures for these are suggested in the converter manual.

The accompanying figure illustrates the flow of the conversion process. Initially, the abstract program may be represented in 8080/8085 or 8086, 8088 assembly language to execute on that respective target machine. The conversion process is porting a source destined for the 8080/8085 to the 86/88 via CONV 86/88.

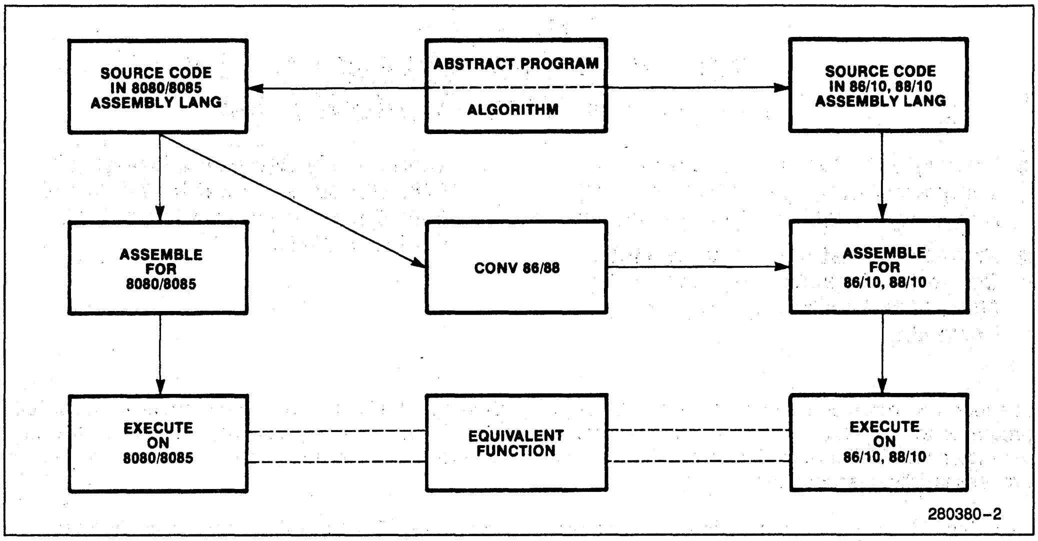


Figure 1. Porting 8080/8085 Source Code to the 8086/10 and 8088/10

LINK 86/88

- **Automatic Combination of Separately Compiled or Assembled 8086, 8088 Programs Into a Relocatable Module**
- **Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Automatic Generation of a Summary Map Giving Results of the LINK 86/88 Process**
- **Abbreviated Control Syntax**
- **Relocatable Modules may be Merged into a Single Module Suitable for Inclusion in a Library**
- **Supports "Incremental" Linking**
- **Supports Type Checking of Public and External Symbols**

LINK 86/88 combines object modules specified in the LINK 86/88 input list into a single output module. LINK 86/88 combines segments from the input modules according to the order in which the modules are listed.

LINK 86/88 will accept libraries and object modules built from PL/M 86/88, ASM 86/88, or any other translator generating Intel's 8086, 8088 Relocatable Object Modules.

Support for incremental linking is provided since an output module produced by LINK 86/88 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK 86/88 supports type checking of PUBLIC and EXTERNAL symbols reporting an error if their types are not consistent.

LINK 86/88 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK 86/88 process and to control the content of the output module.

LINK 86/88 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC 86/88 and enter final testing with much of the work accomplished.

LIB 86/88

- **LIB 86/88 is a Library Manager Program which Allows You to:**

  Create Specially Formatted Files to Contain Libraries of Object Modules

  Maintain These Libraries by Adding or Deleting Modules

  Print a Listing of the Modules and Public Symbols in a Library File
- **Libraries Can be Used as Input to LINK 86/88 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program LIB 86/88 creates and maintains files containing object modules. The operation of LIB 86/88 is controlled by commands to indicate which operation LIB 86/88 is to perform. The commands are:

CREATE:	creates an empty library file
ADD:	adds object modules to a library file
DELETE:	deletes modules from a library file
LIST:	lists the module directory of library files
EXIT:	terminates the LIB 86 program and returns control to ISIS-II

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

LOC 86/88

- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Lengths, and Debug Symbols and their Addresses**
- **Extensive Capability to Manipulate the Order and Placement of Segments in 8086, 8088 Memory**
- **Abbreviated Control Syntax**
- **Automatic and Independent Relocation of Segments. Segments May Be Relocated to Best Match Users Memory Configuration**
- **Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC 86/88 converts relative addresses in an input module to absolute addresses. LOC 86/88 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC 86/88 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC 86/88 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program on the Intellec development system and then simply relocate the object code to suit your application.

OH 86/88

- **Converts an 8086, 8088 Absolute Object Module to Symbolic Hexadecimal Format**
- **Facilitates Preparing a File for Later Loading by a Symbolic Hexadecimal Loader, such as the iSBC™ Monitor SDK-86 Loader, or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The OH 86/88 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary to format a module for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or Universal PROM Mapper. The conversion may also be made to put the module in a more readable format than can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC 86/88 is in absolute format.

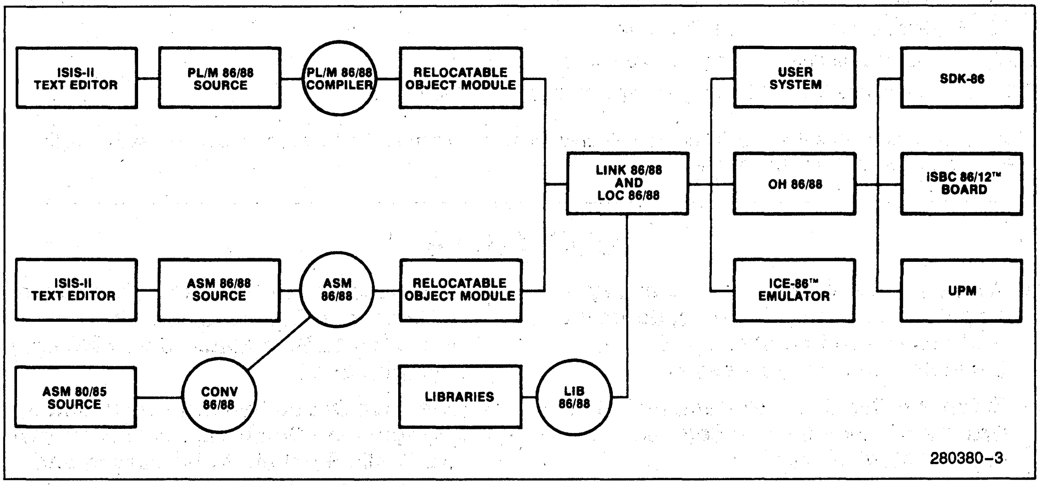


Figure 2. 8086, 8088 Software Development Cycle

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems
Intel Personal Development System

Documentation

PL/M-86 Programming Manual

ISIS-II PL/M-86 Compiler Operator's Manual

MCS-86 User's Manual

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users

MCS-86 Macro Assembly Language Reference Manual

MCS-86 Macro Assembler Operating Instructions for ISIS-II Users

MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users

Universal PROM Programmer User's Manual

ORDERING INFORMATION

8086, 8088 Software Development Packages for Series II:

Part No.	Description
MDS-308*	Assembler and Utilities Package
MDS-311*	PL/M compiler, Assembler, and Utilities Package

All Packages Require Software Licenses

SUPPORT:

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

8086/8088/80186/80188 SOFTWARE PACKAGES

8086 Software Development Package

- **Macro Assembler with Complete System Development Capability for 8086 Designs**
- **Complete Set of Utilities for Object Module Management and Program Linkage**

FORTRAN 8086/8088/80186/80188 Software Package

- **Features High-Level Language Support for Floating-Point Calculation, Transcendentals, Interrupt Procedures, and Run-Time Exception Handling**
- **Meets ANSI FORTRAN 77 Subset Language Specifications**
- **Supports Complex Data Types**

PASCAL 8086/8088/80186/80188 Software Package

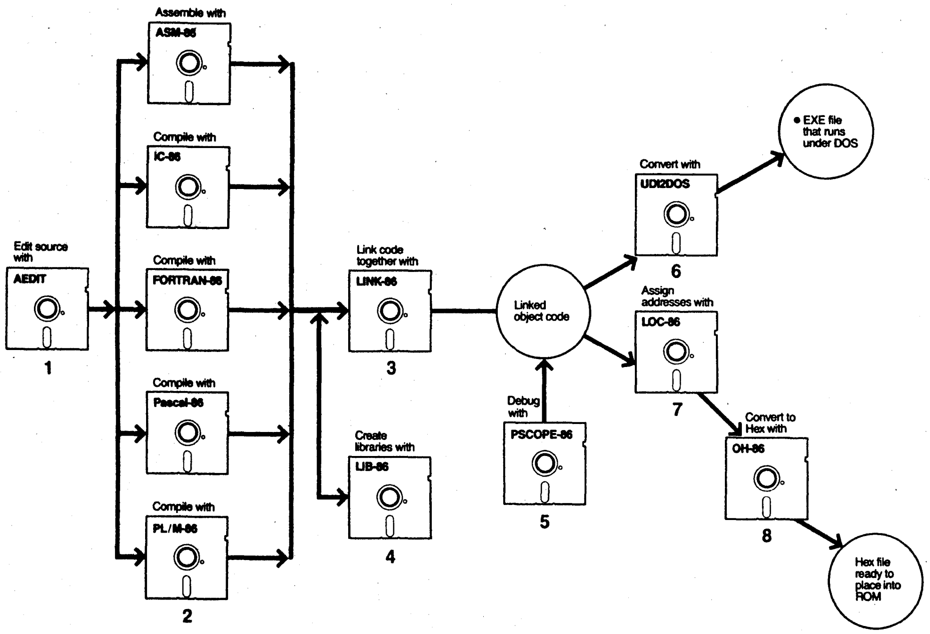
- **Object Compatible and Linkable with PL/M 8086/8088, ASM 8086/8088 and FORTRAN 86/88**
- **Supports Large Array Operation**

PL/M 8086/8088/80186/80188 Software Package

- **Advanced Structured System Implementation Language for Algorithm Development**
- **Easy-to-Learn Block-Structured Language Encourages Program Modularity**

iC-86 Compiler for the 8086

- **Implements Full C Language**
- **Produces High Density Code Rivaling Assembler**



210689-6

Figure 1. Program modules compiled with any of the 8086 languages may be linked together. Each language is compatible with Intel's debug tools and is available hosted on a selection of industry standard systems.

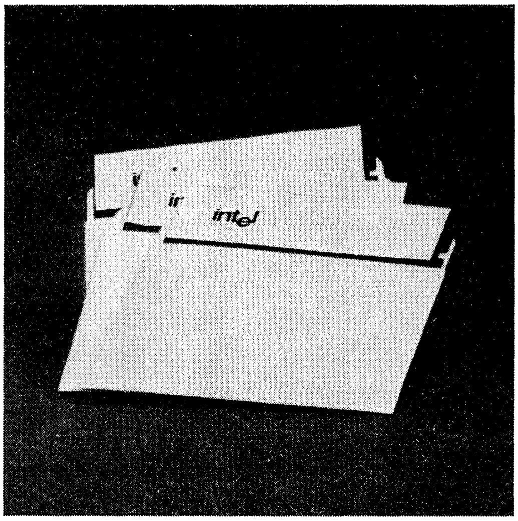
November 1986
Order Number: 210689-007

8086 SOFTWARE DEVELOPMENT PACKAGE

- **Complete System Development Capability for High-Performance 8086 Applications**
- **Macro Assembler for Machine-Level Programming**
- **System Utilities for Program Linkage and Relocation**
- **Package Supports Program Development with PLM-86, Pascal-86, FORTAN 86, & IC 86**
- **Available on a Choice of Hosts**

The 8086 Software Development package contains a macro assembler, a program linker (for linking separately compiled modules together, a system locator, library manager, an object to hex code converter, and a conversion utility to create DOS executable files.

All the utilities in the Software Development Package run on the Intel Microcomputer Development Systems (Series III/Series IV) as well as the IBM PC XT/AT DEC VAX† Minicomputer under the VMS† Operating System, and Intel systems 86/3XX under iRMX™86, and Intel System 286/3XX under iRMX™286.



210689-7

†VAX, VMS are trademarks of Digital Equipment Corporation.

8086/8088/80186/80188 MACRO ASSEMBLER

- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086/8088/80186/80188 Object Modules, Generated by Intel 8086 Compilers**
- **Powerful and Flexible Text Macro Facility with Three Macro Listings Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **"Strongly Typed' Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as "STRUCTURES" and "RECORDS"**
- **Over 120 Detailed and Fully Documented Error Messages**

ASM-86 is the "high-level" macro assembler for the 8086/8088/80186/80188 assembly language. ASM-86 translates symbolic 8086/8088/80186/80188 assembly language mnemonics into 8086/8088/80186/80188 relocatable object code.

ASM-86 should be used where maximum code efficiency and hardware control is needed. The 8086/8088/80186/80188 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 8086/8088/80186/80188 machine instructions. ASM-86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

ASM-86 offers many features normally found only in high-level languages. The 8086/8088/80186/80188 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

LINK-86

- **Automatic Combination of 8086 Programs Separately Translated Using Intel Compilers or Assemblers into Relocatable Object Module**
- **Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References**
- **Extensive Debug Symbol Manipulation, allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Automatic Generation of a Summary Map Giving Results of the LINK-86 Process**
- **Abbreviated Control Syntax**
- **Relocatable Modules May Be Merged into a Single Module Suitable for Inclusion in a Library**
- **Supports "Incremental" Linking**
- **Supports Type Checking of Public and External Symbols**

LINK-86 combines object modules specified in the LINK-86 input list into a single output module. LINK-86 combines segments from the input modules according to the order in which the modules are listed.

LINK-86 will accept libraries and object modules built from any Intel translator generating 8086 Relocatable Object Modules.

Support for incremental linking is provided since an output module produced by LINK-86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK-86 supports type checking of PUBLIC and EXTERNAL symbols reporting a warning if their types are not consistant.

LINK-86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK-86 process and to control the content of the output module.

LINK-86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC-86 and enter final testing with much of the work accomplished.

LOC-86

- **Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Length, and Debug Symbols and Their Addresses**
- **Abbreviated Control Syntax**
- **Segments May be Relocated to Best Match Users Memory Configuration**
- **Extensive Debug Symbol Manipulation Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC-86 converts relative addresses in an input module in 8086/8088/80186/80188 object module format to absolute addresses. LOC-86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC-86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC-86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program and then simply relocate the object code to suit your application.

LIB-86

- **LIB-86 is a Library Manager Program which Allows You to:**
  - **—Create Specifically Formatted Files to Contain Libraries of Object Modules**
  - **—Maintain These Libraries by Adding or Deleting Modules**
  - **—Print a Listing of the Modules and Public Symbols in a Library File**
- **Libraries Can be Used as Input to LINK-86 which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program LIB-86 creates and maintains files containing object modules. The operation of LIB-86 is controlled by commands to indicate which operation LIB-86 is to perform. The commands are:

CREATE:	creates an empty library file
ADD:	adds object modules to a library file
DELETE:	deletes modules from a library file
LIST:	lists the module directory of library files
EXIT:	terminates the LIB-86 program and returns control to VMS

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

OH-86

- **Converts an 8086/8088/80186/80188 Absolute Object Module to Symbolic Hexadecimal Format**
- **Facilitates Preparing a File for Loading by Symbolic Hexadecimal Loader (e.g. iSBC™ Monitor SDK-86 Loader), or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The OH-86 utility converts an 8086/8088 absolute object module to the hexadecimal format. This conversion may be necessary for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or the Universal PROM Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute form; the output from LOC-86 is in absolute format.

SPECIFICATIONS

Documentation Package

ASM-86 Assembly Language Reference Manual

8086/87/88 Macro Assembler Operating Instructions

iAPX 86 Family Utilities User's Guide

Support Available

Software Updates, Subscription Service, Hotline Support

ORDERING INFORMATION

Order Code	Operating Environment
I86ASM86	Intel Series III/Series IV
D86ASM86	IBM PC XT/AT running PC DOS Version 3.0 or later
VVSASM86	VAX†/VMS†
MVVSASM86	MICROVAX†/VMS†
R86ASM86	Intel 86/3XX Systems running: iRMX™ 86
R286ASM286	Intel 286/3XX Systems running: iRMX™ 286

†MICROVAX, VAX, VMS are trademarks of Digital Equipment Corporation.

*IBM, AT are registered trademarks of International Business Machines Corporation.

FORTRAN 8086/8088/80186/80188 SOFTWARE PACKAGE

- **Features High-Level Language Support for Floating-Point Calculations, Transcendentals, Interrupt Procedures, and Run-Time Exception Handling**
- **Meets ANSI FORTRAN 77 Subset Language Specifications**
- **Supports 8086/20, 8088/20 Numeric Data Processor for Fast and Efficient Execution of Numeric Instructions**
- **Uses REALMATH Floating-Point Standard for Consistent and Reliable Results**
- **Supports Arrays Larger Than 64K**
- **Unlimited User Program Symbols**
- **Offers Upward Compatibility with FORTRAN 80**
- **Provides FORTRAN Run-Time Support for 8086, 8088, 80186, 80188-Based Design**
- **Provides Users Ability to do Formatted and Unformatted I/O with Sequential or Direct Access Methods**
- **I²ICE™ Symbolic Debugging Fully Supported**
- **PSCOPE Source Level Debugging Fully Supported**
- **Supports Complex Data Types**
- **Choice of Industry Standard Hosts**

FORTRAN 8086/8088/80186/80188 meets the ANSI FORTRAN 77 Language Subset Specification and includes many features of the full standard. Therefore, the user is assured of portability of most existing ANS FORTRAN programs and of full portability from other computer systems with an ANS FORTRAN 77 Compiler.

FORTRAN 8086/8088/80186/80188 is available to run on the Intel Microcomputer Development Systems (Series III/Series IV) as well as the IBM PC XT/AT running PC DOS Version 3.0 or later, Digital Equipment VAX†/VMS† and Intel System 86/3XX running iRMX™ 86 operating system.

FORTRAN 86/88/186/188 is one of a complete family of compatible programming languages for 8086, 8088, 80186, 80188 development: PL/M, Pascal, FORTRAN, C, and Assembler. Therefore, users may choose the language best suited for a specific problem solution.

---

†VAX, VMS are trademarks of Digital Equipment Corporation.

*IBM, AT are registered trademarks of International Business Machines Corporation.

FEATURES

Extensive High-Level Language Numeric Processing Support

Single (32-bit), double (64-bit), and double extended precision (80-bit) complex (two 32-bit), and double complex (two 64-bit) floating-point data types

REALMATH Proposed IEEE Floating-Point Standard) for consistent and reliable results

Full support for all other data types: integer, logical, character

Ability to use hardware (8086/20, 8088/20 Numeric Data Processor) or software (simulator) floating-point support chosen at link time

ANS FORTRAN 77 Standard

Intel® Microprocessor Support

FORTRAN 8086/8088/80186/80188 language features support of 8086/20, 8088/20 Numeric Data Processor

Compiler generates in-line iAPX 8086/20, 8088/20 Numeric Data Processor object code for floating-point arithmetic (See Figure 2)

Intrinsics allow user to control iAPX 8086/20, 8088/20 Numeric Data processor

8086, 8088, 80186, 80188 architectural advantages used for indexing and character-string handling

Symbolic debugging of application using ICE emulators

Source level debugging using PSCOPE

FLOATING-POINT-STATEMENT

```
      TEMPER = (PRESS - VOLUM / QUEK) - 3.45 / (PRESS - VOLUM / QUEK
     & - (PRESS - VOLUM / QUEK) * (PRESS - VOLUM / QUEK)
```

OBJECT CODE GENERATED

```
 Intel FORTRAN 8086 Compiler
      8086/20, 8088/20          ASSEMBLER MNEMONICS
        MACHINE-CODE                                         ; STATEMENT # 2
0013  9BD9060C00           FLD        VOLUM
0018  9BD8360000           FDIV       QUEK
001D  9BD82E0800           FSUBR      PRESS
0022  9BDDD1               FST        TOS+1H
0025  9B2ED83E0000         FDIVR      CS:@CONST
002B  9BD9C9               FXCHG      TOS+1H
002E  9BDDD2               FST        TOS+2H
0031  9BDEE9               FSUBRP
0034  9BD9C1               FLD        TOS+1H
0037  9BD8C8               FMUL       TOS
003A  9BDDC2               FFREE      TOS+2H
003D  9BDEE1               FSUBP
0040  9BD91E0400           FSTP       TEMPER
0045  9B                   WAIT
```

Figure 2. Object code generated by FORTRAN 8086/8088/80186/80188 for a floating-point calculation using 8086/20, 8088/20 Numeric Processor.

Microprocessor Application Support

— Direct byte- or word-oriented port I/O
— Reentrant procedures
— Interrupt procedures

BENEFITS

FORTRAN 8086/8088/80186/80188 provides a means of developing application software for the Intel 8086/8088/80186/80188 products lines in a familiar, widely accepted, and industry-standard programming language. FORTRAN 8086/8088/80186/80188 will greatly enhance the user's ability to provide cost-effective software development for Intel microprocessors as illustrated by the following:

Early Project Completion

FORTRAN is an industry-standard, high-level numerics processing language. FORTRAN programmers can use FORTRAN 8086/8088/80186/80188 on microprocessor projects with little retraining. Existing FORTRAN software can be compiled with FORTRAN 8086/8088/80186/80188 and programs developed in FORTRAN 8086/8088/80186/80188 can run on other computers with ANS FORTRAN 77 with little or no change. Libraries of mathematical programs using ANS 77 standards may be compiled with FORTRAN 8086/8088/80186/80188.

Application Object Code Portability for a Processor Family

FORTRAN 8086/8088/80186/80188 modules "talk" to the resident Intellec development operating system using Intel's standard interface for all development-system software. This allows an application developed under the ISIS-II operating system to execute on iRMX/86, or a user-supplied operating system by linking in the iRMX/86 or other appropriate interface library. A standard logical-record interface enables communication with non-standard I/O devices.

Comprehensive, Reliable and Efficient Numeric Processing

The unique combination of FORTRAN 8086/8088, 8086/20, 8088/20 Numeric Data processor, and REALMATH (Proposed IEEE Floating-Point Standard) provide universal consistency in results of numeric computations and efficient object code generation.

SPECIFICATIONS

Documentation Package

FORTRAN 86/88/186/188 User's Guide

ORDERING INFORMATION

Order Code	Operating Environment
I86FOR86	Intel Series III/Series IV
D86FOR86	IBM PC XT/AT running PC DOS Version 3.0 or later
R86FOR86	Intel System 86/3XX running iRMX 86
VVS	For 86 VAX/VMS 4.3 and later

SUPPORT AVAILABLE

Software updates, Subscription Service, Hotline Support.

PASCAL 8086/8088/80186/80188 SOFTWARE PACKAGE

- Choice of Industry Standard Hosts
- Object Compatible and Linkable with PL/M 8086/8088, ASM 8086/8088, C8086/8088 and FORTRAN 8086/8088
- I2ICE™ Symbolic Debugging Fully Supported
- PSCOPE Source Level Dubugging Fully Supported
- Implements REALMATH for Consistent and Reliable Results
- Supports Large Array Operation
- Unlimited User Program Symbols
- Supports 8086/20, 8088/20 Numeric Data Processors
- Strict Implementation of ISO Standard Pascal
- Useful Extensions Essential for Microcomputer Applications
- Separate Compilation with Type-Checking Enforced Between Pascal Modules
- Compiler Option to Support Full Run-Time Range-Checking

PASCAL 8086/8088/80186/80188 conforms to and implements the ISO Draft Proposed PASCAL standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The PASCAL 8086/8088/80186/80188 compiler runs on Series III and Series IV Microcomputer Development Systems, as well as the IBM* XT/AT* running PC DOS Version 3.0 or later, Digital Equipment VAX/VMS†, and Intel System 8086/3XX running iRMX™ 86.

A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs as an alternate to the development system environment. Program modules compiled under PASCAL 8086/8088/80186/80188 are compatible and linkable with modules written in PL/M 8086/8088/80186/80188, ASM 8086/8088/80186/80188, C86 or FORTRAN 8086/8088/80186/80188. With a complete family of compatible programming languages for the iAPX 8086/8088/80186/80188 one can implement each module in the language most appropriate to the task at hand.

PASCAL 8086/8088/80186/80188 object modules contain symbol and type information for program debugging using ICE emulators and PSCOPE source language debugger. For final production version, the compiler can remove this extra information and code.

†VAX, VMS are trademarks of Digital Equipment Corporation.

FEATURES

Includes all the language features of Jensen & Wirth Pascal as defined in the ISO Draft Proposed Pascal Standard.

Supports required extensions for microcomputer applications.
— Interrupt handling
— Direct port I/O

Separate compilation extensions allow:
— Modular decomposition of large programs
— Linkage with other Pascal modules as well as PL/M 8086/8088/80186/80188, ASM 8086/8088/80186/80188, C86 and FORTRAN 8086/8088/80186/80188
— Enforcement of type-checking at LINK-time

Supports numerous compiler options to control the compilation process, to INCLUDE files, flag non-standard Pascal statements and others to control program listing and object modules.

Utilizes the IEEE standard for Floating-Point Arithmetic (the Intel REALMATH standard) for arithmetic operations.

Well-defined and documented run-time operating system interfaces allow the user to execute the applications under user-designed operations systems.

Predefined type extensions allow:
— Create precision in read, integer, and unsigned calculations.
— Means to check 8087 errors
— Circumvention of rigid type checking on calls to non-Pascal routines

BENEFITS

Provides a standard Pascal for 8086/8088/80186/80188 based applications.
— Pascal has gained wide acceptance as a portable application language for microcomputer applications
— It is being taught in many colleges and universities around the world
— It is easy to learn, originally intended as a vehicle for teaching computer programming
— Improves maintainability: Type mechanism is both strictly enforced and user extendable
— Few machine specific language constructs

Strict implementation of the proposed ISO standard for Pascal aids portability of application programs. A compile time option checks conformance to the standard making it easy to write conforming programs.

PASCAL 8086/8088/80186/80188 extensions via predefined procedures for interrupt handling and direct port I/O make it possible to code an entire application in Pascal without compromising portability.

Standard Intel REALMATH is easy to use and provides reliable results, consistent with other Intel languages and other implementations of the IEEE proposed Floating-Point standard.

Provides run-time support for co-processors. All real-type arithmetic is performed on the 86/20 numeric data processor unit or software emulator. Run-time library routines, common between Pascal and other Intel languages (such as FORTRAN), permit efficient and consistently accurate results.

Extended relocation and linkage support allows the user to link Pascal program modules with routines written in other languages for certain parts of the program. For example, real-time or hardware dependent routines written in ASM 8086/8088/80186/80188 or PL/M 8086/8088/80186/80188 can be linked to Pascal routines, further extending the user's ability to write structured and modular programs.

PASCAL 8086/8088/80186/80188 programs "talk" to the resident operating system using Intel's standard interface for translated programs. This allows users to replace the development operating system by their own operating systems in the final application.

PASCAL 8086/8088 takes full advantage of 8086/8088/80186/80188 high level language architecture to generate efficient machine code.

Compiler options can be used to control the program listings and object modules. While debugging, the user may generate additional information such as the symbol record information required and useful for debugging using PSCOPE or ICE emulation. After debugging, the production version may be streamlined by removing this additional information.

SPECIFICATIONS

ORDERING INFORMATION

Ordering Code	Operating Environment
I86PAS86	Intel Series III/ Series IV
D86PAS86	IBM PC XT/AT running PC DOS Version 3.0 or later
R86PAS86	Intel System 86/3XX running iRMX™ 86
VVSPAS86	VAX/VMS
MVVPAS86	MICROVAX/VMS

Documentation Package

PASCAL 86 User's Guide

SUPPORT

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

PL/M 8086/8088/80186/80188 Software Package

- Systems Programming Language for the 8086/8088/80186/80188 Processors
- Language is Upward Compatible from PL/M 80, Assuring MCS®-80/85 Design Portability
- Advanced Structured System Implementation Language for Algorithm Development
- Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard
- Easy-to-Learn Block-Structured Language Encourages Program Modularity
- Improved Compiler Performance Now Supports More User Symbols and Faster Compilation Speeds
- Produces Relocatable Object Code Which Is Linkable to All Other 8086 Object Modules
- Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization
- Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors
- Resident on Choice of Hosts
- I^2ICE Symbolic Debugging Fully Supported
- PSCOPE Source Level Debugging Fully Supported

PL/M 8086 is an advanced, structured, high-level systems programming language. The PL/M 8086 compiler was created specifically for performing software development for the Intel 8086, 8088, 80186 and 80188 Microprocessors. PL/M was designed so that program statements naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 8086 compiler efficiently converts free-form PL/M language statements into machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-up maintenance costs for the user.

PL/M 8086 is available to run on the Intellec® Microcomputer Development Systems (Series III/Series IV) as well as the IBM PC XT/AT, DEC VAX†/VMS†, and Intel System 8086/3XX running iRMX™ 86.

†VAX, VMS are trademarks of Digital Equipment Corporation.

FEATURES

Major features of the Intel PL/M 8086 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 8086 implementation of a block structure allows the use of REENTRANT (recursive) procedures, which are especially useful in system design.

Language Compatibility

PL/M 8086 object modules are compatible with object modules generated by all other 8086 translators. This means that PL/M programs may be linked to programs written in any other 8086 language.

Object modules are compatible with In-Circuit Emulators; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 8086 Language is upward compatible with PL/M 80, so that application programs may be easily ported to run on the 8086.

Supports Seven Data Types

PL/M makes use of seven data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

- Byte: 8-bit unsigned number
- Word: 16-bit unsigned number
- DWORD: 32-bit unsigned number
- Integer: 16-bit signed number
- Read: 32-bit floating point number
- Pointer: 16-bit or 32-bit memory address indicator
- Selector: 16-bit base portion of a pointer

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These help the user to organize data into logical groups.

- Array: Indexed list of same type of data elements
- Structure: Named collection of same or different type data elements
- Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the 8086/20 or 8088/20 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or Emulator takes place at linktime, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 8086 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for handling interrupts. A procedure may be defined with the INTERRUPT attribute, and the compiler willl automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to save and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET$INTERRUPT, the function retuning an INTERRUPT$PTR, and the PL/M statement CAUSE$INTERRUPT all add flexibility to user programs involving interrupt and handling.

Compiler Controls

Including several that have been mentioned, the PL/M 8086 compiler offers more than 25 controls that facilitate such features as:

— Conditional compilation

— Including additional PL/M source files from disk

— Corresponding assembly language code in the listing file

— Setting overflow conditions for run-time handling

Segmentation Control

The PL/M 8086 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64 KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 8086 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions

— "Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block

— Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreachable code

— Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations

Error Checking

The PL/M 8086 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation errors is provided by the compiler.

```
M:DO; /* Beginning of module */
        SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) PUBLIC;
                  DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER,
        /* Parameters:
             PTR is pointer to first record.
             COUNT is number of records to be sorted.
             RECSIZE is number of bytes in each record—max is 128.
             KEYINDEX is byte position within each record of a BYTE scalar
                        to be used as sort key. */
                 DECLARE RECORD BASED PTR (1) BYTE,
                                    CURRENT (128) BYTE,
                                    (I, J) INTEGER;
        SORT:    DO J=1 TO COUNT-1;
                      CALL MOVB(@RECORD(J*RECSIZE), @CURRENT, RECSIZE);
                      I=J;
        FIND:         DO WHILE I>0
                                  AND RECORD((I-1)*RECSIZE+KEYINDEX)
                                  >CURRENT(KEYINDEX);
                          CALL MOVB(@RECORD((I-1)*RECSIZE),
                                          @RECORD(I*RECSIZE),
                                          RECSIZE);
                          I=I-1;
                      END FIND;
                      CALL MOVB(@CURRENT, @RECORD(I*RECSIZE), RECSIZE);
                 END SORT;
             END SORTPROC;
END M;           /*End of module*/
```

PUBLIC and EXTERNAL attributes promote program modularity.

"Based" Variables allow manipulation of external data by passing the base of the data structure (a pointer). This minimizes the STACK space used for parameter passing, and the execution time to perform many STACK operations.

The "AT" operator returns the *address* of a variable, instead of its contents. This is very useful in passing pointers for based variables.

One of several PL/M built-in procedures for string manipulation.

210689-5

Figure 3. Sample PL/M 8086 Program

BENEFITS

PL/M 8086 is designed to be an efficient, cost-effective solution to the special requirements of 8086 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Cost-Effective Alternative to Assembly Language

PL/M 8086 programs are code efficient. PL/M 8086 combines all of the benefits of a high-level language (ease of use, high productivity) with the ability to access the 8086 architecture. Consequently, for the development of systems software, PL/M 8086 is the cost-effective alternative to assembly language programming.

Low Learning Effort

PL/M is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 8086, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because fewer programming resources are required for a given programmed function.

Increased Reliability

PL/M 8086 is designed to aid in the development of reliable software (PL/M 8086 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

SPECIFICATIONS

Documentation Package

PL/M-8086 User's Guide for 8086-based Development Systems

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.

ORDERING INFORMATION

Order Code	Operating Environment
I86PLM86	Intel Series III/Series IV
D86PLM86	IBM PC XT/AT running PCDOS Version 3.0 or later
R86PLM86	Intel System 8086/3XX running iRMX™ 86
WSPLM86	VAX/VMS
MVVSPLM86	MICROVAX/VMS

iC-86
C COMPILER FOR THE 8086

- **Implements Full C Language**
- **Produces High Density Code Rivaling Assembler**
- **Supports Intel Object Module Format (OMF)**
- **Runs under the Intel UDI on Intel Development Systems and iRMX™ 86**
- **Available for the VAX/VMS* Operating System**
- **Supports PSCOPE-86 and I²ICE™**
- **Supports Both Small and Large Models of Computation**
- **Supports IEEE Floating Point Math with 8087 Coprocessor**
- **Supports Bit Fields**
- **Supports Full Standard I/O Library (STDIO)**
- **Written in C**

The C Programming Language was originally designed in 1972 and has become increasingly popular as a systems development language. C is not a "very high level" language and is not tied to any specific application area. Although it is used for writing operation systems, it has been used equally well to write numerical, text-processing and data base programs. C combines the flexibility and programming speed of a higher level language with the efficiency and control of assembly language.

Intel iC-86 brings the full power of the C programming language to 8086 and 8088 based microprocessor systems.

Intel iC-86 supports the full C language as described in the Kernighan and Ritchie book, "The C Programming Language", (Prentice-Hall, 1978). Also included are the latest enhancements to the C language: structure assignments, functions taking structure arguments and returning structures, and the "void" and "enum" data types.

C is rapidly becoming the standard microprocessor system implementation language because it provides:

1. the ability to manipulate the fundamental objects of the machine (including machine addresses) as easily as assembly language.
2. the power and speed of a structured language supporting a large number of data types, storage classes, expressions and statements,
3. processor independence (most programs developed for other processors can be easily transported to the 8086), and
4. code that rivals assembly language in efficiency

INTEL iC-86 COMPILER DESCRIPTION

The iC-86 compiler operates in four phases: preprocessor, parser, code generator, and optimizer. The preprocessor phase interprets directives in C source code, including conditional compilations (# define). The parser phase converts the C program into an intermediate free form and does all syntactic and semantic error checking. The code generator phase converts the parser's output into an efficient intermediate binary code, performs constant folding, and features an extremely efficient register allocator, ensuring high quality code. The optimizer phase converts the output of the code generator into relocatable Intel Object Module Format (OMF) code, without creating an intermediate assembly file. Optionally, the iC-86 compiler can produce a symbolic assembly like file. The iC-86 optimizer eliminates common code, eliminates redundant loads and stores, and resolves span dependencies (shortens branches) within a program.

The iC-86 runtime library consists of a number of functions which the C programmer can call. The runtime system includes the standard I/O library (STDIO), conversion routines, routines for manipulating strings, special routines to perform functions not available on the 8086 (32-bit arithmetic and emulated floating point), and (where appropriate) routines for interfacing with the operating system.

iC-86 uses Intel's linker and locator and generates debug records for symbols and lines on request, permitting access to Intel's PSCOPE AND I^{2}ICE™ to aid in program testing.

FEATURES

Support for Small and Large Models

Intel iC-86 supports both the SMALL and LARGE modes of segmentation. A SMALL model program can have up to 64K bytes of code and 64K bytes of data, with all pointers occupying two bytes. Because two byte pointers permit the generation of highly compact and efficient code, this model is recommended for programs that can meet the size restrictions. The LARGE segmentation model is used by programs that require access to the full addressing space of the 8086/8088 processors. In this model, each source file generates a distinct pair of code and data segments of up to 64K bytes in length. All pointers are four bytes long.

Preprocessor Directives

#define—defines a macro

#include—includes code outside of the program source file

#if—conditionally includes or excludes code

Other preprocessor directives include #undef, #ifdef, #ifndef, #else, #endif, and #line.

Statements

The C language supports a variety of statements:

Conditionals: IF, IF-ELSE

Loops: WHILE, DO-WHILE, FOR

Selection of cases: SWITCH, CASE, DEFAULT

Exit from a function: RETURN

Loop control: CONTINUE, BREAK

Branching: GOTO

Expressions and Operators

The C language includes a rich set of expressions and operators.

Primary expression: invoke functions, select elements from arrays, and extract fields from structures or unions

Arithmetic operators: add, subtract, multiply, divide, modulus

Relational operators: greater than, greater than or equal, less than, less than or equal, not equal

Unary operators: indirect through a pointer, compute an address, logical negation, ones complement, provide the size in bytes of an operand.

Logical operators: AND, OR

Bitwise operators: AND, exclusive OR, inclusive OR, bitwise complement

Data Types and Storage Classes

Data in C is described by its type and storage class. The type determines its representation and use, and the storage class determines its lifetime, scope, and storage allocation. The following data types are fully supported by iC-86.

char

an 8-bit signed integer

int

a 16-bit signed integer

short

same as int (on the 8086)

long

a 32-bit signed integer

unsigned

a modifier for integer data types (char, int, short, and long) which doubles the positive range of values

float

a 32-bit floating point number which utilizes the 8087 or a software floating point library

double

a 64-bit floating point number

void

a special type that cannot be used as an operand in expressions; normally used for functions called only for effect (to prevent their use in contexts where a value is required).

enum

an enumerated data type

These fundamental data types may be used to create other data types including: arrays, functions, structures, pointers, and unions.

The storage classes available in iC-86 include:

register

suggests that a variable be kept in a machine register, often enhancing code density and speed

extern

a variable defined outside of the function where it is declared; retaining its value throughout the entire program and accessible to other modules

auto

a local variable, created when a block of code is entered and discarded when the block is exited

static

a local variable that retains its value until the termination of the entire program

typedef

defines a new data type name from existing data types

BENEFITS

Faster Compilation

Intel iC-86 compiles C programs substantially faster than standard C compilers because it produces Intel OMF code directly, eliminating the traditional intermediate process of generating an assembly file.

Portability of Code

Because Intel iC-86 supports the STDIO and produces Intel OMF code, programs developed on a variety of machines can easily be transported to the 8086.

Rapid Program Development

Intel iC-86 provides the programmer with detailed error messages and access to PSCOPE-86 and I^{2}ICE to speed program development.

Full Manipulation of the 8086

Intel iC-86 enables the programmer to utilize features of the C language to control bit fields, pointers, addresses and register allocation, taking full advantage of the fundamental concepts of the 8086.

SPECIFICATIONS

Operating Environment

The iC-86 compiler runs host resident on both the Intel Series III Microcomputer Development System under ISIS-II and on the System 86/330 under the iRMX™ 86 operating system. iC-86 can also run as a cross compiler on a VAX 11/780 computer under the VMS operating system 128K bytes of User Memory is required on all versions. The PC DOS Operating Environment is also supported. Specify desired version when ordering.

Required Hardware

Development System Version

— Intellec® Microcomputer Development System; Series III or Series IV

— Dual Diskette Drives, Single or Double Density

— System Console; CRT or Hardcopy Interactive Device

iRMX 86 version:

— Any 8086/8088, iSBC® 86/88, iTPS 86/XXX, or SYS 86/3XX based system capable of running the iRMX 86 Operating System

VAX version:

— Digital Equipment Corporation VAX 11/780 or compatible computer

PC DOS version:

— PC XT or AT using PC DOS 3.0 or later

Optional Hardware

ISIS-II version:

— ICE-86, I²ICE-86

iRMX 86 version:

— Numeric Data Processors for support of the REALMATH standard

VAX version:

— None

Required Software

ISIS-II version:

— ISIS-II Diskette Operating System

— Series III or Series IV Operating

iRMX 86 version:

— iRMX 86 Realtime Multiprogramming Operating System

— iRMX 860 Utilities Package

VAX version:

— VMS Operating System

PC DOS version:

— PC DOS Release 3.0 or later Operating System

Optional Software

Development System Version:

— None

iRMX 86 version:

— None

VAX version:

— MDS*-384 Kit-Mainframe Link for distributed development, or iMDX-394 Asynchronous Communications Link.

— VAX 8086/8088/80186 MACRO Assembler and utilities package (iMDX-341VX)

Documentation Package

The C Programming Language by Kernighan and Ritchie (1978 Prentice-Hall)

iC-86 User Manual

Shipping Media

Development System Version:

— Two single and one double density ISIS-II format 8″ diskettes, one 5¼″ Series IV Format

iRMX 86 version:

— Double Density iRMX 86 format 8″ diskette

— Double Density iRMX 86 format 5¼″ diskette

VAX version:

— 1600 bpi, 9 track Magnetic tape

PC DOS version:

— 5¼″ PC DOS format diskette

ORDERING INFORMATION

Order Code	Description
I86C86	iC-86 Compiler for ISIS-II
R86C86	iC-86 Compiler for iRMX 86
iMDX-347	iC-86 Cross Compiler for VAX/VMS
D86C86	iC-86 Cross Compiler for PC DOS

Intel Software License required

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

VAX, VMS are registered trademarks of Digital Equipment Corporation.

intel®

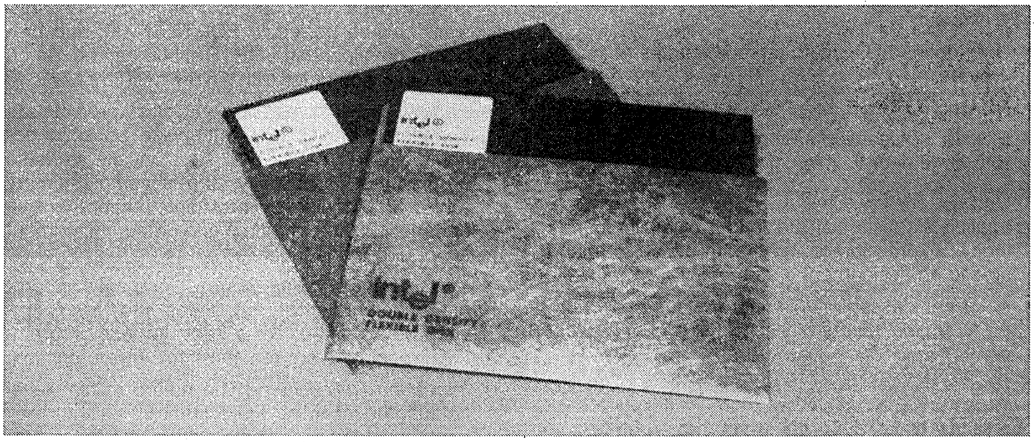
FORTRAN 80 8080/8085 ANS FORTRAN 77 INTELLEC® RESIDENT COMPILER

- Meets ANS FORTRAN 77 Subset Language Specification Plus Adds Intel® Microcprocessor Extensions
- Supports Intel Floating Point Standard with the FORTRAN 80 Software Routines, the iSBC-310™ High Speed Mathematics Board, or the iSBC-332™ Math Multimodule
- Executes on Intellect Microcomputer Development System, Intellec Series II Microcomputer Development System and Personal Development System
- Supports Full Symbolic Debugging with ICE-80™ and ICE-85™
- Produces Relocatable and Linkable Object Code Compatible with Resident PL/M 80 and 8080/8085 Macro Assembler
- Provides Optional Run-Time Library to Execute in RMX-80™ Environment
- Has Well Defined I/O Interface for Configuration with User-Supplied Drivers

FORTRAN 80 is a computer industry-standard, high-level programming language and compiler that translates FORTRAN statements into relocatable object modules. When the object modules are linked together and located into absolute program modules, they are suitable for execution on Intel 8080/8085 Microprocessors, iSBC-80 OEM Computer Systems, Intellec Microcomputer Development Systems and Personal Development Systems. FORTRAN 80 meets the ANS FORTRAN 77 Language Subset Specification.(1) In addition, extensions designed specifically for microprocessor applications are included. The compiler operates on the Intellec Microcomputer Development System and Personal Development System under the ISIS-II Disk Operating System and produces efficient relocatable object modules that are compatible for linkage with PL/M 80 and 8080/8085 Macro Assembler modules.

The ANS FORTRAN 77 language specification offers many powerful extensions to the FORTRAN language that are especially well suited to Intel 8080/8085 Microprocessor software development. Because FORTRAN 80 conforms to the ANS FORTRAN 77 standard, the user is assured of compatibility with existing FORTRAN software that meets the standard as well as a guarantee of upward compatibility to other computer systems supporting an ANS FORTRAN 77 Compiler.

(1) ANSI X3J3/90



400610–1

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

November 1986
Order Number: 400610-001

FORTRAN 80 LANGUAGE FEATURES

Major ANS FORTRAN 77 features supported by the Intel FORTRAN 80 Programming Language include:

- Structured Programming is supported with the IF ... THEN ... ELSE IF ... ELSE ... END IF constructs.
- CHARACTER data type permits alphanumeric data to be handled as strings rather than characters stored in array elements.
- Full I/O capabilities include:

— Sequential and Direct Access files

— Error handling facilities

— Formatted, Free-formatted, and Unformatted data representation

— Internal (in-memory) file units provide capability to format and reformat data in internal memory buffers

— List directed formatting

— Supports arrays of up to seven dimensions.

— Supports logical operators

.EQV.	— Logical equivalence
.NEQV.	— Logical nonequivalence

Major extensions to FORTRAN 77 in Intel FORTRAN-80 include:

- Direct 8080/8085 port I/O supported by intrinsic subroutines.
- Binary and Hexadecimal integer constants.
- Well defined interface to FORTRAN-80 I/O statements (READ, OPEN, etc.) allowing easy use of user-supplied I/O drivers.
- User-defined INTEGER storage lengths of 1, 2 or 4 bytes.
- User-defined LOGICAL storage lengths of 1, 2 or 4 bytes.
- REAL STORAGE lengths of 4 bytes.
- Bitwise Boolean operations using logical operators on integer values.
- Hollerith data constants.
- Implicit extension of the length of an integer or logical expression to the length of the left-hand side in an assignment statement.
- A format descriptor to suppress carriage return on a terminal output device at the end of the record.

FORTRAN 80 COMPILER FEATURES

- Supports multiple compilation units in single source file.
- Optional Assembly Language code listing.
- Comprehensive cross-reference, symbol attribute and error listing.
- Compiler controls and directives are compatible with other Intel language translators.
- Optional Reentrancy.
- User-defined default storage lengths.
- Optional FORTRAN 66 Do Loop semantics.
- Source files may be prepared in free format.
- The INCLUDE control permits specified source files to be combined into a compilation unit at compile time.
- Tranparent interface for software and hardware floating point support, allowing either to be chosen at time of linking.

FORTRAN 80 BENEFITS

FORTRAN 80 provides a means of developing application software for Intel MCS-80/85 products in a familiar, widely accepted, and computer industry-standardized programming language. FORTRAN 80 will greatly enhance the user's ability to provide cost-effective solutions to software development for Intel microcoprocessors as illustrated by the following:

- *Completely Complementary to Existing Intel Software Design Tools* — Object modules are linkable with new or existing Assembly Language and PL/M Modules.
- *Incremental Runtime Library Support* — Runtime overhead is limited only to facilities required by the program.
- *Low Learning Effort* — FORTRAN 80, like PL/M, is easy to learn and use. Existing FORTRAN software can be ported to FORTRAN 80, and programs developed in FORTRAN 80 can be run on any other computer with ANS FORTRAN 77.
- *Earlier Project Completion* — Critical projects are completed earlier than otherwise possible because FORTRAN 80 will substantially increase programmer productivity, and is complementary to PL/M Modules by providing comprehensive arithmetic, I/O formatting, and data management support in the language.
- *Lower Development Cost* — Increases in programmer productivity translates into lower software development costs because less programming resources are required for a given function.
- *Increased Reliability* — The nature of high-level languages, including FORTRAN 80, is that they lend themselves to simple statements of the program algorithm. This substantially reduces the risk of costly errors in systems that have already reached production status.

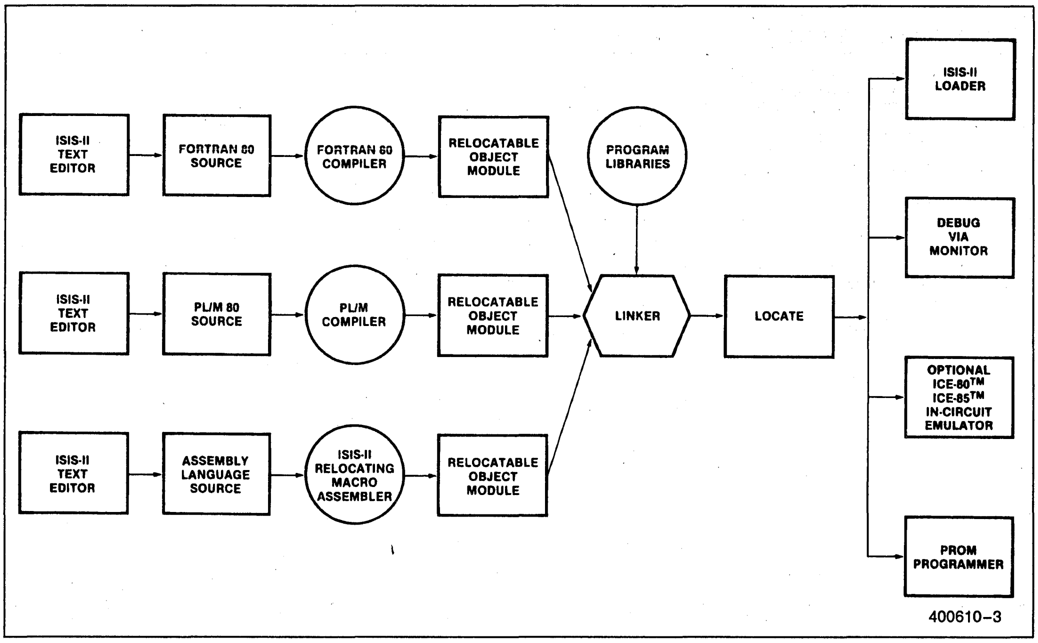
SAMPLE FORTRAN-80 SOURCE PROGRAM LISTING

```
*    **  THIS PROGRAM IS AN EXAMPLE OF ISIS-II FORTRAN-80 THAT
*    **  CONVERTS TEMPERATURE BETWEEN CELSIUS AND FARENHEIT

      PROGRAM CONVRT

      CHARACTER*1 CHOICE, SCALE

      PRINT 100
*     **  ENTER CONVERSION SCALE (C OR F)
10    PRINT 200
      READ (5,300) SCALE

        IF (SCALE .EQ. 'C')
     +     THEN
              PRINT 400
*             **  ENTER THE NUMBER OF DEGREES FARENHEIT
              READ (5,*) DEGF
              DEGC = 5./9.*(DEGF-32)
*             **  PRINT THE ANSWER
              WRITE (6,500) DEGF,DEGC
*             **  RUN AGAIN?
20            PRINT 600
              READ (5,300) CHOICE
                 IF (CHOICE .EQ. 'Y')
     +              THEN
                       GOTO 10
                 ELSE IF (CHOICE .EQ. 'N')
     +              THEN
                       CALL EXIT
                 ELSE
                    GOTO 20
                 END IF
        ELSE IF (SCALE .EQ. 'F')
     +     THEN
*             **  CONVERT FROM FARENHEIT TO CELSIUS
              PRINT 700
              READ (5,*) DEGC
              DEGF = 9./5.*DEGC+32.
*             **  PRINT THE ANSWER
              WRITE (6,800) DEGC,DEGF
              GOTO 20
        ELSE
*          **  NOT A VALID ENTRY FOR THE SCALE
           WRITE (6,900) SCALE
           GOTO 10
        END IF
100   FORMAT(' TEMPERATURE CONVERSION PROGRAM',//,
     +' TYPE C FOR FARENHEIT TO CELSIUS OR',/,
     +' TYPE F FOR CELSIUS TO FARENHEIT',//)
200   FORMAT(/,' CONVERSION? ',$)
300   FORMAT(A1)
400   FORMAT(/,'ENTER DEGREES FARENHEIT: ',$)
500   FORMAT(/,F7.2,' DEGREES FARENHEIT = ',F7.2,' DEGREES CELSIUS')
600   FORMAT(/,' AGAIN (Y OR N)? ',$)
700   FORMAT(/,' ENTER DEGREES CELSIUS: ',$)
800   FORMAT(/,F7.2,' DEGREES CELSIUS = ',F7.2,' DEGREES FARENHEIT',/)
900   FORMAT(/,1H ,A1,' NOT A VALID CHOICE - TRY AGAIN!',/)
      END
```

400610-2

- *Easier Enhancements and Maintenance* — Like PL/M, program modules written in FORTRAN 80 are easier to read and understand than assembly language. This means it is easier to enhance and maintain FORTRAN 80 programs as system capabilities expand and future products are developed.
- *Comprehensive, Yet Simple Project Development* — The Intellec Microcomputer Development System and Personal Development System, with the 8080/8085 Macro Assembler, PL/M 80 and FORTRAN 80 are the most comprehensive software design facilities available for the Intel MCS-80/85 Microprocessor family. This reduces development time and cost because expensive (and remote) timesharing or large computers are not required.

The FORTRAN 80 Compiler is an efficient, multiphase compiler that accepts source programs, translates them into relocatable object code, and produces requested listings. After compilation, the object program may be linked to other modules, located to a specific area of memory, then executed. The diagram shown below illustrates a program development cycle where the program consists of modules created by FORTRAN 80, PL/M 80 and the 8080/8085 Macro Assembler.



SPECIFICATIONS

OPERATING ENVIRONMENT

Required Hardware:

1. Intel Microcomputer Development Systems
2. Personal Development Systems

DOCUMENTATION PACKAGE

FORTRAN-80 Programming Manual

ISIS-II FORTRAN-80 Compiler Operator's Manual

FORTRAN-80 Programming Reference Card

ORDERING INFORMATION

Part Number	Description
Model MDS-301	FORTRAN 80 Compiler for Intellect Microcomputer Development Systems.

Requires Software License.

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

PASCAL 80 SOFTWARE PACKAGE

- Offers a Superset of Standard Pascal
- Provides High Structured Language with Powerful Data Type Definitions to Suit Applications
- Compiles Pascal Source Code into Intermediate Code to Optimize Execution Speed and Storage
- Executes Compiler and Interprets the Intermediate Code on Intellec® Microcomputer Development Systems
- Can Call Routines Written in PL/M 80, FORTRAN 80, or 8080/8085 Macro Assembler
- Provides a Utility to Produce Relocatable Object Modules Compatible with Other Intel® Languages
- Allows Modular Breakdown of Large Programs and Separate Compilation of Individual Modules
- Gives Application Control Over Run-Time Errors by Providing User-Declared Error Procedures

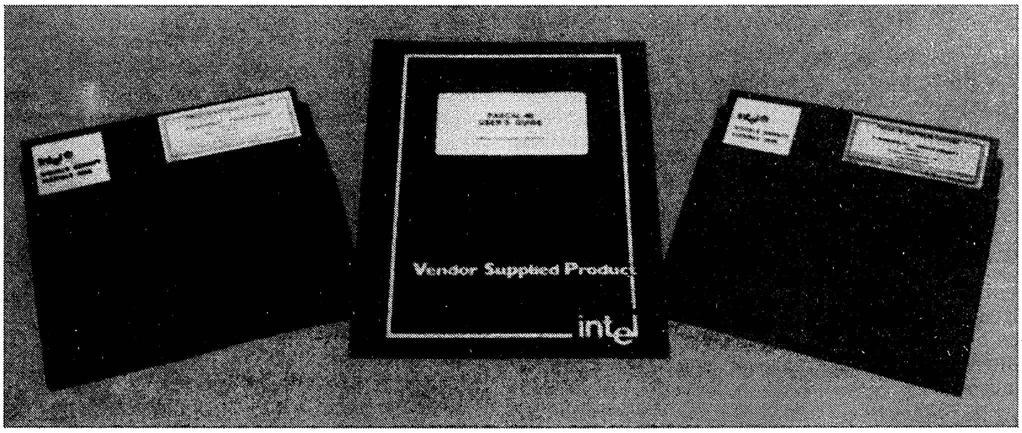
PASCAL 80 Software Package consists of a compiler and an interactive Run-Time System designed to provide the Pascal programming language as a software development tool for Intellec Development System Users.

Pascal is a highly-structured, block-oriented programming language that is now gaining wide acceptance as a powerful software development tool. Its rigid structure encourages and enforces good programming techniques which, combined with a high level of readability, helps produce more reliable software.

Standard Intel development tools, such as CREDIT editor can be used to create and modify Pascal source programs. The compiler compiles this source and creates a P-Code file. The Run-Time System executes this P-Code in an interpretive manner under ISIS-II.

*Pascal language as defined in *PASCAL User Manual and Report,* Second Edition, Kathleen Jenson and Niklaus Wirth.

**MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.



280379-1

LANGUAGE FEATURES

Data Structures

Pascal allows the user to define labels, constants, data types, variables, procedures, and functions.

Variable Types

Variables can be defined according to the following system-defined data types: boolean, integer, real, character, array, record, string, set, file, and pointer.

User-Defined Types

New types can be defined by the user for added flexibility.

File Handling Procedures

Pascal provides procedures to allow a user's program to interface with the ISIS-II file manager. Routines provided are: RESET, REWRITE, CLOSE, PUT, GET, SEEK, and PAGE.

Input/Output Procedures

Routines are provided to interact with the console or an ISIS file. These procedures are: READ, WRITE, READLN, WRITELN, plus BUFFER and BLOCK Read and Write.

Dynamic Memory Allocation

The procedures NEW, MARK, and RELEASE allow the user to obtain and release memory space at run-time for dynamically allocating variable storage.

String Handling

Pascal provides powerful tools for defining and manipulating strings and character arrays. These facilities enable concatenation of strings, character and pattern scans, insertion, deletion, and pointer manipulation.

Recursion

Pascal allows a PROCEDURE defintion to include a call to itself, a powerful construct in many mathematical algorithms.

PROGRAM TRACING FACILITY

The PASCAL 80 System incorporates a program tracing facility which allows for selectively monitoring the execution of a Pascal program. When the TRACE flag is set, the line number of each program statement being executed is output to the console.

The TRACE flag may be manipulated in two ways:

— The TRACEON command (of the Run-Time System) will set the flag, and the TRACEOFF command will reset the flag.

— Pressing the Interrupt 4 switch on the Intellec System front panel will toggle the TRACE flag; i.e., the flag will be set if it was reset, and vice-versa.

COMPILER DIRECTIVES (PARTIAL LIST)

Compiler Command Line Directives

NO LIST

No list file is produced; used for fast compilation of "clean" programs.

NOCODE

No code file is produced; used for syntax error checking.

ERRLIST

List file is limited to only those Pascal lines that contain errors, along with the error messages produced.

LIST (file-name)

Specifies the name of the list file.

CODE (file-name)

Specifies the name of the code file.

NOECHO

Error lines are echoed on the console unless this directive is specified.

Embedded Compiler Directives

$C text

Causes text to appear in code file (allows for comments, copyrights, etc.).

$I+

Causes checking for I/O completion after each I/O transfer. Failure results in a run-time error. ($I− causes no checking, and no errors on I/O failure.)

$R+

Causes Range Checking to occur, so that an out-of-range value causes a run-time error. ($R− suppresses generation of code for Range Checking.)

$O+

Causes the compiler to operate in overlay mode. Overlays allows less source code to reside in memory. ($O− causes no overlays, which decreases compile time, since there are fewer disk accesses.)

$T+

Causes the compiler to generate tracing instructions to be used by the TRACE facility. ($T− suppresses tracing instructions.)

BENEFITS

Brings Pascal to Intellec Microcomputer Development Systems:

— Pascal is a block-structured, highly-readable programming language, suitable for a wide-range of applications.

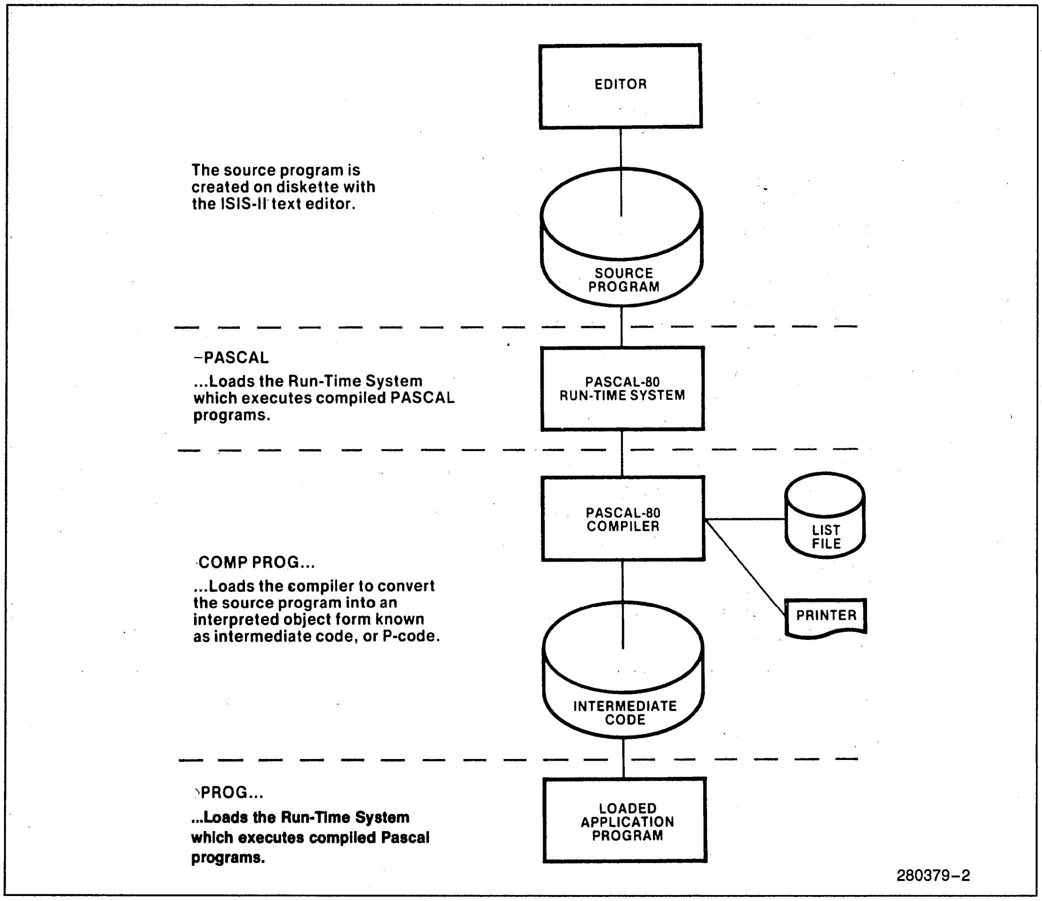

EDITOR
The source program is created on diskette with the ISIS-II text editor.
SOURCE PROGRAM
-PASCAL
...Loads the Run-Time System which executes compiled PASCAL programs.
PASCAL-80 RUN-TIME SYSTEM
PASCAL-80 COMPILER
LIST FILE
PRINTER
COMP PROG...
...Loads the compiler to convert the source program into an interpreted object form known as intermediate code, or P-code.
INTERMEDIATE CODE
-PROG...
...Loads the Run-Time System which executes compiled Pascal programs.
LOADED APPLICATION PROGRAM
280379-2

Figure 1. Program Development Cycle

— Pascal is being acclaimed as the programming language of the future; it is being taught in many colleges and universities around the country.

— PASCAL 80 Run-Time System provides great ease in programming formatted I/O operations.

PASCAL 80 provides a portable language for application programs running under ISIS-II.

PASCAL 80 can be used to evaluate complicated algorithms using a natural language.

PASCAL 80 compiler generates intermediate Pseudo-code.

— P-code is optimized for speed and storage space.

— P-code is approximately 50% to 70% smaller than corresponding machine code.

— P-code is machine independent, providing code portability to any CPU.

Makes the Intellec Development System a more valuable tool. Extension of software support to include Pascal makes software development and resource management more flexible.

Table 1. Sample Program Listing Showing Nesting Levels

```
                          BUFFER.PAS Program Listing
Line Seg Proc Lev Disp
  1   1   1         1 program example;
  2   1   1         3
  3   1   1         3 { Example using bufferread and bufferwrite with break characters }
  4   1   1         3
  5   1   1         3 var buffer: string;
  6   1   1        44     disk_storage: file;
  7   1   1        64     break: char;
  8   1   1        65     new_len, len: integer;
  9   1   1        67     buff_array: packed array[0..80]of char;
 10   1   1       108
 11   1   1    0    0 begin
 12   1   1    1    0   rewrite (disk_storage, 'data');
 13   1   1    1   27   writeln('Input a line of text:');
 14   1   1    1   68   readln(buffer);
 15   1   1    1   87   len :=bufferwrite(disk_storage, buffer[1],length(buffer));
 16   1   1    1  109   repeat
 17   1   1    2  109     reset(disk_storage);
 18   1   1    2  116     writeln;writeln;
 19   1   1    2  132     write('Input break char [cntrl Z to stop]:');
 20   1   1    2  179     readln(break);
 21   1   1    2  197     if not eof(input)then
 22   1   1    3  208       begin
 23   1   1    4  208         new_len: =bufferread(disk_storage,buff_array,len,ord(break));
 24   1   1    4  226         writeln('The buffer read:');
 25   1   1    4  262         writeln(copy(buffer,1,abs(new_len)));
 26   1   1    4  292         writeln('Length:',abs(new_len):0);
 27   1   1    4  331         if new_len < 0 then writeln('(Break char not found)');
 28   1   1    3  378     end;
 29   1   1    1  378   until eof(input);
 30   1   1    0  388 end.
```

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intellec Microcomputer Development Systems
- Model 800 (Series II, Series III, Series IV)
- Intel Personal Development System

REQUIRED SOFTWARE

ISIS-II Diskette Operating System
- Single-or Double-Density

OPTIONAL SOFTWARE

ISIS-II CREDIT™ (CRT-Based Text Editor)

Documentation Package

PASCAL 80 User's Guide (9801015-01)

PASCAL User Manual and Report, Second Edition, Kathleen Jensen and Niklaus Wirth

Shipping Media

Flexible Diskettes
- Single- and Double-Density

ORDERING INFORMATION

Part Number	Description
MDS-381**	PASCAL 80 Software Package

Requires Software License

SUPPORT CATEGORY: Level D

PL/M 80 HIGH LEVEL PROGRAMMING LANGUAGE

- **Provides Resident Operation on Intellec® Microcomputer Development System and Intellec® Series II Microcomputer Development Systems and Personal Development Systems (PDS)**
- **Produces Relocatable and Linkable Object Code**
- **Sophisticated Code Optimization Reduces Application Memory Requirements**
- **Speeds Project Completion with Increased Programmer Productivity**
- **Cuts Software Development and Maintenance Costs**
- **Improves Product Reliability with Simplified Language and Consequent Error Reduction**
- **Eases Enhancement as System Capabilities Expand**

The PL/M 80 High Level Programming Language Intellec Resident Compiler is an advanced, high level programming language for Intel 8080 and 8085 microprocessors, iSBC-80 OEM computer systems, and Intellec microcomputer development systems. PL/M has been substantially enhanced since its introduction in 1973 and has become one of the most effective and powerful microprocessor systems implementation tools available. It is easy to learn, facilitates rapid program development and debugging, and significantly reduces maintenance costs. PL/M is an algorithmic language in which program statements naturally express the algorithm to be programmed, thus freeing programmers to concentrate on system development rather than assembly language details (such as register allocation, meanings of assembler mnemonics, etc.). The PL/M compiler efficiently converts free-form PL/M programs into equivalent 8080/8085 instructions. Substantially fewer PL/M statements are necessary for a given application than would be using assembly language or machine code. Since PL/M programs are problem oriented and thus more compact, programming in PL/M results in a high degree of productivity during development efforts, resulting in significant cost reduction in software development and maintenance for the user.

210327–1

MDS™ is a registered trademark of Mohawk Data Sciences Corporation.

November 1986
Order Number: 210327-002

FUNCTIONAL DESCRIPTION

The PL/M compiler is an efficient multiphase compiler that accepts source programs, translates them into object code, and produces requested listings. After compilation, the object program may be first linked to other modules, then located to a specific area of memory, and finally executed. The diagram shown in Figure 1 illustrates a program development cycle where the program consists of three modules: PL/M, FORTRAN, and assembly language. A typical PL/M compiler procedure is shown in Table 1.

Features

Major features of the Intel PL/M 80 compiler and programming language include:

Resident Operation—on Intellec microcomputer development systems eliminates the need for a large in-house computer or costly timesharing system.

Object Code Generation—of relocatable and linkable object codes permits PL/M program development and debugging in small modules, which may be easily linked with other modules and/or library routines to form a complete application.

Extensive Code Optimization—including compile time arithmetic, constant subscript resolution, and common subexpression elimination, results in generation of short, efficient CPU instruction sequences.

Symbolic Debugging—fully supported in the PL/M compiler and ICE-85 in-circuit emulators.

Compile Time Options—includes general listing format commands, symbol table listing, cross reference listing, and "innerlist" of generated assembly language instructions.

Block Structure—aids in utilization of structured programming techniques.

Access—provided by high level PL/M statements to hardware resources (interrupt systems, absolute addresses, CPU input/output ports).

Data Definition—enables complex data structures to be defined at a high level.

Re-entrant Procedures—may be specified as a user option.

Benefits

PL/M is designed to be an efficient, cost-effective solution to the special requirements of microcomputer software development as illustrated by the following benefits of PL/M use:

Low Learning Effort—even for the novice programmer, because PL/M is easier to learn.

Earlier Project Completion—on critical projects because PL/M substantially increases programmer productivity while reducing program development time.

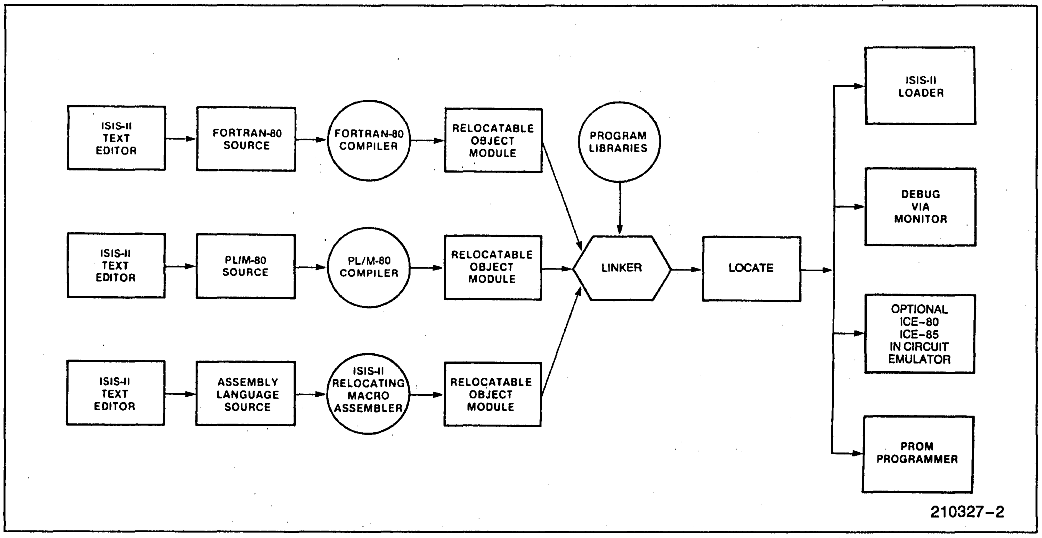


Figure 1. Program Development Cycle Block Diagram

Lower Development Cost—because increased programmer productivity requiring less programming resources for a given function translates into lower software development costs.

Increased Reliability—because of PL/M's use of simple statements in the program algorithm, which are easier to correct and thus substantially reduce the risk of costly errors in systems that have already reached full production status.

Easier Enhancement and Maintenance—because programs written in PL/M are easier to read and easier to understand than assembly language, and thus are easier to enhance and maintain as system capabilities expand and future products are developed.

Simpler Project Development—because the Intellect microcomputer development system with resident PL/M 80 is all that is needed for developing and debugging software for 8080 and 8085 microcomputers, and the use of expensive (and remote) timesharing or large computers is consequently not required.

Table 1. PL/M-80 Compiler Sample Factorial Generator Procedure

```
                 $OBJECT(:F1:FACT.OB2)
                 $DEBUG
                 $XREF
                 $TITLE('FACTORIAL GENERATOR — PROCEDURE')
                 $PAGEWIDTH(80)

 1               FACT:
                 DO;

 2   1           DECLARE NUMCH BYTE PUBLIC;

 3   1           FACTORIAL: PROCEDURE (NUM,PTR) PUBLIC;
 4   2              DECLARE NUM BYTE, PTR ADDRESS;
 5   2              DECLARE DIGITS BASED PTR (161) BYTE;
 6   2              DECLARE (I,C,M) BYTE;

 7   2              NUMCH=1; DIGITS(1)=1;
 9   2              DO M=1 TO NUM;
10   3                 C=0;
11   3                 DO I=1 TO NUMCH;
12   4                    DIGITS(I)=DIGITS(I)*M+C;
13   4                    C=DIGITS(I)/10;
14   4                    DIGITS(I)=DIGITS(I) — 10*C;
15   4                 END;

16   3                 IF C<>0 THEN
17   3                 DO;
18   4                    NUMCH=NUMCH+1; DIGITS(NUMCH)=C;
20   4                    C=DIGITS(NUMCH)/10;
21   4                    DIGITS(NUMCH)=DIGITS(NUMCH) — 10*C;
22   4                 END
                    END;

24   2           END FACTORIAL;

25   1           END;                                         210327-3
```

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems (Series II, Series III, Series IV)
Intel Personal Development System

DOCUMENTATION

PL/M 80 Programming Manual
ISIS-II PL/M 80 Compiler Operator's Manual

ORDERING INFORMATION

Product Code	Description
MDS*-PLM	PL/M 80 High Level Language Compiler. Needs Software License.

SUPPORT

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

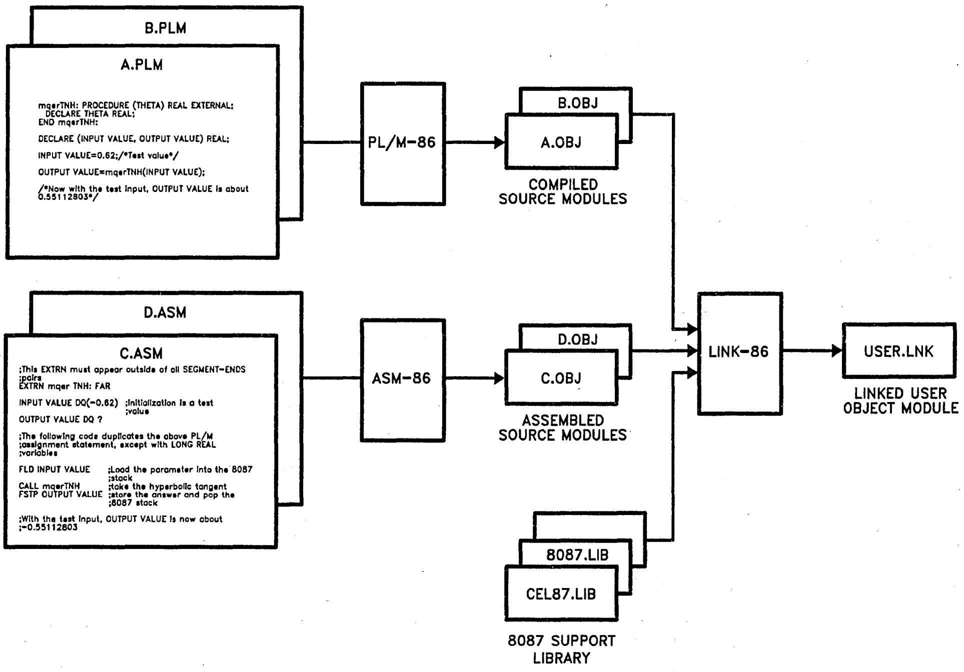
***NOTE:**
MDS is an ordering code only and is not used as a product or trademark.

8087 SUPPORT LIBRARY

- **Library to Support Floating Point Arithmetic in Pascal-86, PL/M-86, FTN-86 and ASM-86**
- **Decimal Conversion Library Supports Binary-Decimal Conversions**
- **Supports Proposed IEEE Floating Point Standard for High Accuracy and Software Portability**
- **Common Elementary Function Library Provides Trigonometric, Logarithmic and Other Useful Functions**
- **Error-Handler Module Simplifies Floating Point Error Recovery**

The 8087 Support Library provides Pascal-86, FORTRAN-86, PL/M-86 and ASM-86 users with numeric data processing capability. With the Library, it is easy for programs to do floating point arithmetic. Programs can bind in library modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. Figure 1 below illustrates how the 8087 Support Library can be bound with PL/M-86 and ASM-86 user code to do this. The 8087 Support Library supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable–the software investment is maintained.

The 8087 Support Library consists of the common elementary function library (CEL87.LIB), the decimal conversion library (DC87.LIB), the emulator interface library E8087.LIB, the error handler module (EH87.LIB) and interface libraries (8087.LIB, NUL87.LIB).



```
mqerTNH: PROCEDURE (THETA) REAL EXTERNAL;
  DECLARE THETA REAL;
END mqerTNH:

DECLARE (INPUT VALUE, OUTPUT VALUE) REAL;

INPUT VALUE=0.62;/*Test value*/

OUTPUT VALUE=mqerTNH(INPUT VALUE);

/*Now with the test Input, OUTPUT VALUE is about
0.55112803*/
```

```
;This EXTRN must appear outside of all SEGMENT-ENDS
;pairs
EXTRN mqer TNH: FAR

INPUT VALUE DQ(-0.62)  ;Initialization is a test
                       ;value
OUTPUT VALUE DQ ?

;The following code duplicates the above PL/M
;assignment statement, except with LONG REAL
;variables

FLD INPUT VALUE       ;Load the parameter into the 8087
                      ;stack
CALL mqerTNH          ;take the hyperbolic tangent
FSTP OUTPUT VALUE     ;store the answer and pop the
                      ;8087 stack

;With the test Input, OUTPUT VALUE is now about
;-0.55112803
```

Figure 1. Use of 8087 Support Library with PL/M-86 and ASM-86

August 1985
Order Number: 231613-001

CEL87.LIB THE COMMON ELEMENTARY FUNCTION LIBRARY

FUNCTIONS

CEL87.LIB contains commonly used floating point functions. It is used along with the 8087 numeric coprocessor. It provides a complete package of elementary functions, giving valid results for all appropriate inputs. Following is a summary of CEL87 functions, grouped by functionality.

Rounding and Truncation Functions:

mqerIEX,	mqerIE2, and mqerIE4. Round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.
mqerIAX,	mqerIA2, mqerIA4. Round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.
mqerICX,	mqerIC2, mqerIC4. Truncate the fractional part of a real input; the answer is real, a 16-bit integer or 32-bit integer, repectively.

Logarithmic and Exponential Functions:

mqerLGD	computes decimal (base 10) logarithms.
mqerLGE	computes natural base (base e) logarithms.
mqerEXP	computes exponentials to the base e.
mqerY2X	computes exponentials to any base.
mqerY12	raises an input real to a 16-bit integer power.
mqerY14	is as mqerY12, except to a 32-bit integer power.
mqerYIS	is as mqerY12, but it accommodates PL/M-286 users.

Trigonometric and Hyperbolic Functions:

mqerSIN,	mqerCOS, mqerTAN compute sine, cosine, and tangent.
mqerASN,	mqerACS, mqerATN compute the corresponding inverse functions.
mqerSNH,	mqerCSH, mqerTNH compute the corresponding hyperbolic functions.
mqerAT2	is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions (of real variables):

mqerDIM	is FORTRAN's positive difference function.
mqerMAX	returns the maximum of two real inputs.
mqerMIN	returns the minimum of two real inputs.
mqerSGH	combines the sign of one input with the magnitude of the other input.
mqerMOD	computes a modulus, retaining the sign of the dividend.
mqerRMD	computes a modulus, giving the value closest to zero.

Complex Number Functions:

mqercCMUL,	and mqercCDIV perform complex multiplication and division of complex numbers.
mqercCPOL	converts complex numbers from rectangular to polar form. mqercCREC converts complex numbers from polar to rectangular form.
mqercCSQR,	and mqercCABS compute the complex square root and real absolute value (magnitude) of a complex number.
mqercCEXP,	and mqercCLGE compute the complex value of e raised to a complex power and the complex natural logarithm (base e) of a complex number.
mqercCSIN,	mqercCCOS, and mqercCTAN compute the complex sine, cosine, and tangent of a complex number.
mqercCASN,	mqercCACS, and mqercCATN compute the complex inverse sine, cosine, and tangent of a complex number.
mqercCSNH,	mqercCCSH, and mqercCTNH compute the complex hyperbolic sine, cosine, and tangent of a complex number.

mqercCACH,	mqercCASH, and mqercCATH compute the comples inverse hyperbolic sine, cosine, and tangent of a complex number.
mqercCC2C,	mqercCR2C, mqercCC2R, mqercCCI2, mqercCCI4, and mqercCCIS return complex values of complex (or real) values raised to complex (real, short integer, or long integer) values.

DC87.LIB THE DECIMAL CONVERSION LIBRARY

DC87.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure mqcBIN_DECLOW accepts a binary number in any of the formats used for the representation of floating point numbers in the 8087. Because there are so many output formats for floating point numbers, mqcBIN_DECLOW does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure mqcDEC_BIN accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure mqcDECLOW_BIN is provided for callers who have already broken the decimal number into its constituent parts.

The procedures mqcLONG_TEMP, mqcSHORT_TEMP, mqcTEMP_LONG, and mqcTEMP_SHORT convert floating point numbers between the longest binary format, TEMP_REAL, and the shorter formats.

EH87.LIB THE ERROR HANDLER LIBRARY

EH87.LIB is a library of five utility procedures for writing trap handlers. Trap handlers are called when an unmasked 8087 error occurs.

The 8087 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 8087 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 8087, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL in your trap handler, you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 8087 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH87.LIB can be accomplished with a single call to FILTER.

8087.LIB, NUL87.LIB, E8087.LIB INTERFACE LIBRARIES

E8087.LIB, 8087.LIB and NUL87.LIB libraries configure a user's application program for his run-time environment; running with the 8087 component or without floating point arithmetic, respectively.

FULL 8087 EMULATOR

The Full 8087 Emulator is a 16-kilobyte object module that is linked to the application program for floating-point operations. Its functionality is identical to the 8087 chip, and is ideal for prototyping and debugging floating-point applications. The Emulator is an alternative to the use of the 8087 chip, although the latter executes floating-point applications up to 100 times faster than an 8086 with the 8087 Emulator. Furthermore, since the 8087 is a "coprocessor," use of the chip will allow many operations to be performed in parallel with the 8086.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III, Series IV)

Documentation Package

8087 Support Library Reference Manual

ORDERING INFORMATION

Part Number	Description
iMDS 319	8087 Support Library

Requires Software License

SUPPORT

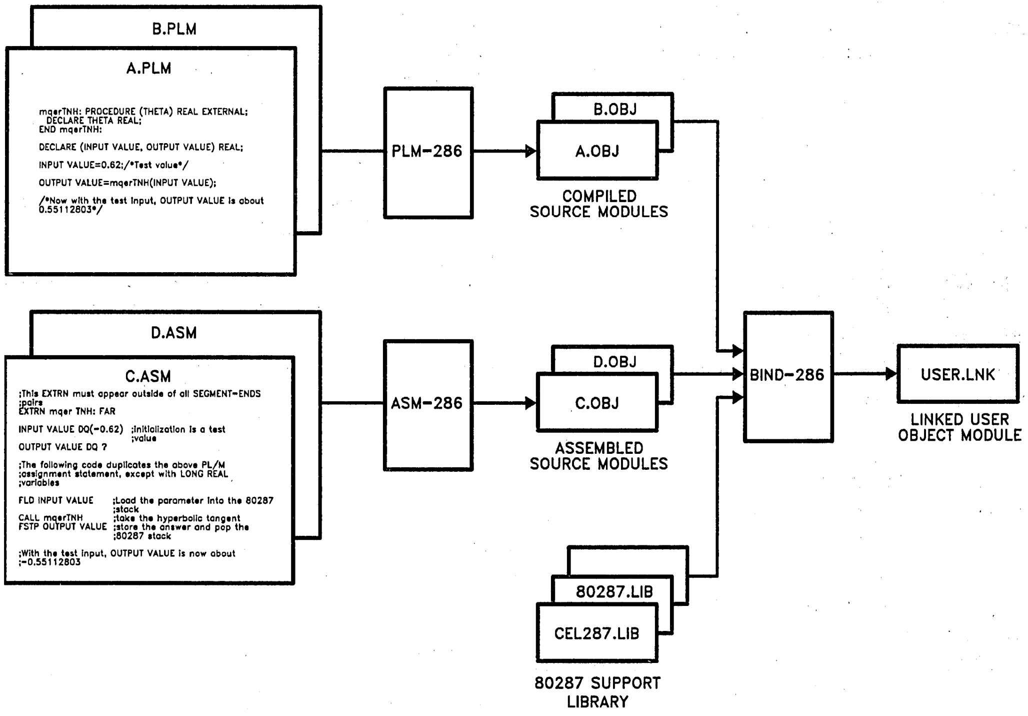
Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

80287 SUPPORT LIBRARY

- **Library to support floating point arithmetic in Pascal-286, PL/M-286 and ASM-286**
- **Decimal conversion library supports binary-decimal conversions**
- **Supports proposed IEEE Floating Point Standard for high accuracy and software portability**
- **Common elementary function library provides trigonometric, logarithmic and other useful functions**
- **Error-handler module simplifies floating point error recovery**

The 80287 Support Library provides Pascal-286, PL/M-286 and ASM-286 users with numeric data processing capability. With the Library, it is easy for programs to do floating point arithmetic. Programs can bind in library modules to do trigonometric, logarithmic and other numeric functions, and the user is guaranteed accurate, reliable results for all appropriate inputs. Figure 1 below illustrates how the 80287 Support Library can be bound with PL/M-286 and ASM-286 user code to do this. The 80287 Support Library supports the proposed IEEE Floating Point Standard. Consequently, by using this Library, the user not only saves software development time, but is guaranteed that the numeric software meets industry standards and is portable–the software investment is maintained.

The 80287 Support Library consists of the common elementary function library (CEL287.LIB), the decimal conversion library (DC287.LIB), the error handler module (EH287.LIB) and interface libraries (80287.LIB, NUL287.LIB).



```
mqerTNH: PROCEDURE (THETA) REAL EXTERNAL;
  DECLARE THETA REAL;
END mqerTNH:

DECLARE (INPUT VALUE, OUTPUT VALUE) REAL;

INPUT VALUE=0.62;/*Test value*/

OUTPUT VALUE=mqerTNH(INPUT VALUE);

/*Now with the test input, OUTPUT VALUE is about
0.55112803*/
```

```
;This EXTRN must appear outside of all SEGMENT-ENDS
;pairs
EXTRN mqer TNH: FAR

INPUT VALUE DQ(-0.62)  ;Initialization is a test
                       ;value
OUTPUT VALUE DQ ?

;The following code duplicates the above PL/M
;assignment statement, except with LONG REAL
;variables

FLD INPUT VALUE      ;Load the parameter into the 80287
                     ;stack
CALL mqerTNH         ;take the hyperbolic tangent
FSTP OUTPUT VALUE    ;store the answer and pop the
                     ;80287 stack

;With the test input, OUTPUT VALUE is now about
;-0.55112803
```

231041-1

Figure 1. Use of 80287 Support Library with PL/M-286 and ASM-286

October 1986
Order Number: 231041-002

CEL287.LIB THE COMMON ELEMENTARY FUNCTION LIBRARY

FUNCTIONS

CEL287.LIB contains commonly used floating point functions. It is used along with the 80287 numeric coprocessor. It provides a complete package of elementary functions, giving valid results for all appropriate inputs. Following is a summary of CEL287 functions, grouped by functionality.

Rounding and Truncation Functions:

mqerIEX,	mqerIE2, and mqerIE4. Round a real number to the nearest integer; to the even integer if there is a tie. The answer returned is real, a 16-bit integer or a 32-bit integer respectively.
mqerIAX,	mqerIA2, mqerIA4. Round a real number to the nearest integer, to the integer away from zero if there is a tie; the answer returned is real, a 16-bit integer or a 32-bit integer, respectively.
mqerICX,	mqerIC2, mqerIC4. Truncate the fractional part of a real input; the answer is real, a 16-bit integer or 32-bit integer, repectively.

Logarithmic and Exponential Functions:

mqerLGD	computes decimal (base 10) logarithms.
mqerLGE	computes natural base (base e) logarithms.
mqerEXP	computes exponentials to the base e.
mqerY2X	computes exponentials to any base.
mqerY12	raises an input real to a 16-bit integer power.
mqerY14	is as mqerY12, except to a 32-bit integer power.
mqerYIS	is as mqerY12, but it accommodates PL/M-286 users.

Trigonometric and Hyperbolic Functions:

mqerSIN,	mqerCOS, mqerTAN compute sine, cosine, and tangent.
mqerASN,	mqerACS, mqerATN compute the corresponding inverse functions.
mqerSNH,	mqerCSH, mqerTNH compute the corresponding hyperbolic functions.
mqerAT2	is a special version of the arc tangent function that accepts rectangular coordinate inputs.

Other Functions (of real variables):

mqerDIM	is FORTRAN's positive difference function.
mqerMAX	returns the maximum of two real inputs.
mqerMIN	returns the minimum of two real inputs.
mqerSGH	combines the sign of one input with the magnitude of the other input.
mqerMOD	computes a modulus, retaining the sign of the dividend.
mqerRMD	computes a modulus, giving the value closest to zero.

Complex Number Functions:

mqercCMUL,	and mqercCDIV perform complex multiplication and division of complex numbers.
mqercCPOL	converts complex numbers from rectangular to polar form. mqercCREC converts complex numbers from polar to rectangular form.
mqercCSQR,	and mqercCABS compute the complex square root and real absolute value (magnitude) of a complex number.
mqercCEXP,	and mqercCLGE compute the complex value of e raised to a complex power and the complex natural logarithm (base e) of a complex number.
mqercCSIN,	mqercCCOS, and mqercCTAN compute the complex sine, cosine, and tangent of a complex number.
mqercCASN,	mqercCACS, and mqercCATN compute the complex inverse sine, cosine, and tangent of a complex number.
mqercCSNH,	mqercCCSH, and mqercCTNH compute the complex hyperbolic sine, cosine, and tangent of a complex number.

Complex Number Functions: (Continued)

mqercCACH, mqercCASH, and mqercCATH compute the comples inverse hyperbolic sine, cosine, and tangent of a complex number.

mqercCC2C, mqercCR2C,mqercCC2R,mqercCCI2, mqercCCI4, and mqercCCIS return complex values of complex (or real) values raised to complex (real, short integer, or long integer) values.

DC287.LIB THE DECIMAL CONVERSION LIBRARY

DC287.LIB is a library of procedures which convert binary representations of floating point numbers and ASCII-encoded string of digits.

The binary-to-decimal procedure mqcBIN__DECLOW accepts a binary number in any of the formats used for the representation of floating point numbers in the 80287. Because there are so many output formats for floating point numbers, mqcBIN__DECLOW does not attempt to provide a finished, formatted text string. Instead, it provides the "building blocks" for you to use to construct the output string which meets your exact format specification.

The decimal-to-binary procedure mqcDEC__BIN accepts a text string which consists of a decimal number with optional sign, decimal point, and/or power-of-ten exponent. It translates the string into the caller's choice of binary formats.

Decimal-to-binary procedure mqcDECLOW__BIN is provided for callers who have already broken the decimal number into its constituent parts.

The procedures mqcLONG__TEMP, mqcSHORT__TEMP, mqcTEMP__LONG, and mqcTEMP__SHORT convert floating point numbers between the longest binary format, TEMP__REAL, and the shorter formats.

EH287.LIB THE ERROR HANDLER LIBRARY

EH287.LIB is a library of five utility procedures for writing trap handlers. Trap handlers are called when an unmasked 80287 error occurs.

The 80287 error reporting mechanism can be used not only to report error conditions, but also to let software implement IEEE standard options not directly supported by the chip. The three such extensions to the 80287 are: normalizing mode, non-trapping not-a-number (NaN), and non-ordered comparison. The utility procedures support these extra features.

DECODE is called near the beginning of the trap handler. It preserves the complete state of the 80287, and also identifies what function called the trap handler, and returns available arguments and/or results. DECODE eliminates much of the effort needed to determine what error caused the trap handler to be called.

NORMAL provides the "normalizing mode" capability for handling the "D" exception. By calling NORMAL in your trap handler, you eliminate the need to write code in your application program which tests for non-normal inputs.

SIEVE provides two capabilities for handling the "I" exception. It implements non-trapping NaN's and non-ordered comparisons. These two IEEE standard features are useful for diagnostic work.

ENCODE is called near the end of the trap handler. It restores the state of the 80287 saved by DECODE, and performs a choice of concluding actions, by either retrying the offending function or returning a specified result.

FILTER calls each of the above four procedures. If your error handler does nothing more than detect fatal errors and implement the features supported by SIEVE and NORMAL, then your interface to EH287.LIB can be accomplished with a single call to FILTER.

80287.LIB, NUL287.LIB INTERFACE LIBRARIES

80287.LIB and NUL287.LIB libraries configure a user's application program for his run-time environment; running with the 80287 component or without floating point arithmetic, respectively.

SPECIFICATIONS

Operating Environment

Intel Microcomputer Development Systems (Series III, Series IV)

Documentation Package

80287 Support Library Reference Manual

Related Software

A 80287 software emulator is available as part of the 8086 software toolbox (iMDX364)

ORDERING INFORMATION

Part Number	Description
iMDX329	80287 Support Library

Requires Software License

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

8051 SOFTWARE PACKAGES

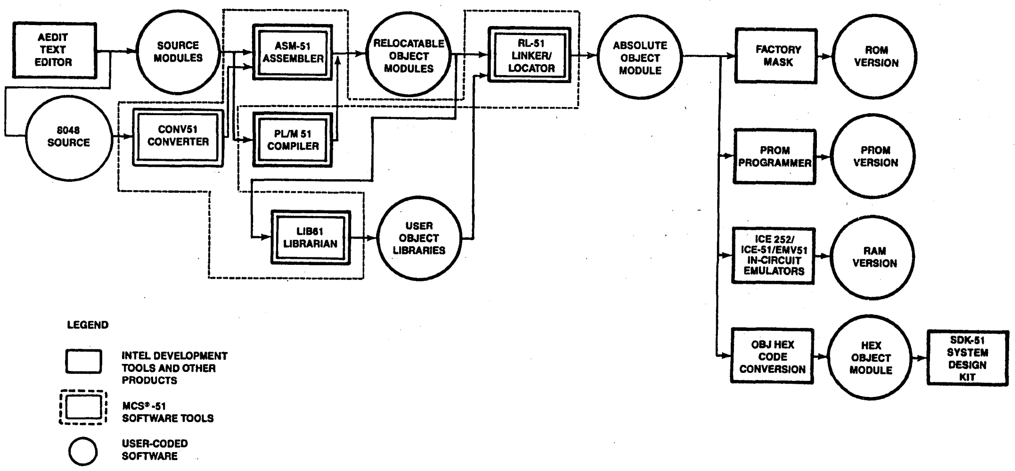
- Choice of hosts: PCDOS 3.0 based IBM* PC XT/AT*, iPDS™ System, Series II, Series III, and Series IV
- Supports all members of the Intel MCS® -51 architecture

PL/M51 Software Package Contains the following:

- PL/M51 Compiler which is designed to support all phases of software implementation
- RL51 Linker and Relocator which enables programmers to develop software in a modular fashion
- LIB51 Librarian which lets programmers create and maintain libraries of software object modules

8051 Software Development Package Contains the following:

- 8051 Macro Assembler which gives symbolic access to 8051 hardware features
- RL51 Linker and Relocator program which links modules generated by the assembler
- CONV51 which enables software written for the MCS® -48 family to be upgraded to run on the 8051
- LIB51 Librarian which lets programmers create and maintain libraries of software object modules



162771-1

Figure 1. MCS® -51 Program Development Process

*IBM and AT are registered trademarks of International Business Machine Corporation.

October 1986
Order Number: 162771-004

PL/M 51 SOFTWARE PACKAGE

- **High-level programming language for the Intel MCS® -51 single-chip microcomputer family**
- **Compatible with PL/M 80 assuring MCS® -80/85 design portability**
- **Enhanced to support boolean processing**
- **Tailored to provide an optimum balance among on-chip RAM usage, code size and code execution time**
- **Produces relocatable object code which is linkable to object modules generated by all other 8051 translators**
- **Allows programmer to have complete control of microcomputer resources**
- **Extends high-level language programming advantages to microcontroller software development**
- **Improved reliability, lower maintenance costs, increased programmer productivity and software portability**
- **Includes the linking and relocating utility and the library manager**
- **Supports all members of the Intel MCS® -51 architecture**

PL/M 51 is a structured, high-level programming language for the Intel MCS-51 family of microcomputers. The PL/M 51 language and compiler have been designed to support the unique software development requirements of the single-chip microcomputer environment. The PL/M language has been enhanced to support Boolean processing and efficient access to the microcomputer functions. New compiler controls allow the programmer complete control over what microcomputer resources are used by PL/M programs.

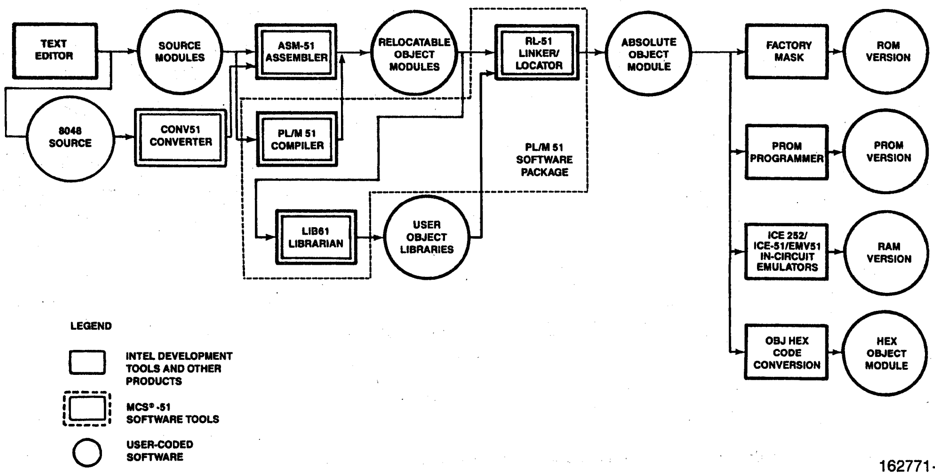
PL/M 51 is largely compatible with PL/M 80 and PL/M 86. A significant proportion of existing PL/M software can be ported to the MCS-51 with modifications to support the MCS-51 architecture. Existing PL/M programmers can start programming for the MCS-51 with a small relearning effort.

PL/M 51 is the high-level alternative to assembly language programming for the MCS-51. When code size and code execution speed are not critical factors, PL/M 51 is the cost-effective approach to developing reliable, maintainable software.

The PL/M 51 compiler has been designed to support efficiently all phases of software implementation with features like a syntax checker, multiple levels of optimization, cross-reference generation and debug record generation.

ICE™ 5100, ICE 51, and EMV51 are available for on-target debugging.

Software available for PC DOS 3.0 based IBM* PC XT/AT* Systems, iPDS™, Series II, Series III and Series IV Systems.



162771-2

Figure 2. PL/M51 Software Package

PL/M 51 COMPILER

FEATURES

Major features of the Intel PL/M 51 compiler and programming language include:

Structured Programming

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure, for example).

Language Compatibility

PL/M 51 object modules are compatible with object modules generated by all other MCS-51 translators. This means that PL/M programs may be linked to programs written in any other MCS-51 language.

Object modules are compatible with In-Circuit Emulators and Emulation Vehicles for MCS-51 processors: the DEBUG compiler control provides these tools with symbolic debugging capabilities.

Supports Three Data Types

PL/M makes use of three data types for various applications. These data types range from one to sixteen bits and facilitate various arithmetic, logic, and address functions:

— Bit: a binary digit

— Byte: 8-bit unsigned number or,

— Word: 16-bit unsigned number.

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

PL/M 51 supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

— Array: Indexed list of same type data elements

— Structure: Named collection of same or different type data elements

— Combinations of Both: Arrays of structures or structures of arrays.

Interrupt Handling

A procedure may be defined with the INTERRUPT attribute. The compiler will generate code to save and restore the processor status, for execution of the user-defined interrupt handler routines.

Compiler Controls

The PL/M 51 compiler offers controls that facilitate such features as:

— Including additional PL/M 51 source files from disk

— Cross-reference

— Corresponding assembly language code in the listing file

Program Addressing Control

The PL/M 51 compiler takes full advantage of program addressing with the ROM (SMALL/MEDIUM/LARGE) control. Programs with less than 2 KB code space can use the SMALL or MEDIUM option to generate optimum addressing instructions. Larger programs can address over the full 64 KB range.

Code Optimization

The PL/M 51 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; "Strength reductions" (a shift left rather than multiply by 2)

— Machine code optimizations; elimination of superfluous branches

— Automatic overlaying of on-chip RAM variables

— Register history: an off-chip variable will not be reloaded if its value is available in a register.

Error Checking

The PL/M 51 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation error messages is provided by the compiler and user's guide.

BENEFITS

PL/M 51 is designed to be an efficient, cost-effective solution to the special requirements of MCS-51 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 51 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 51, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 51 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL51 LINKER AND RELOCATOR

- **Links modules generated by the assembler and the PL/M compiler**
- **Locates the linked object to absolute memory locations**
- **Enables modular programming of software-efficient program development**
- **Modular programs are easy to understand, maintainable and reliable**

The MCS-51 linker and relocator (RL51) is a utility which enables MCS-51 programmers to develop software in a modular fashion. The utility resolves all references between modules and assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The total number of allowed symbols in user-developed software is very large because the assembler number of symbols' limit applies only per module, not to the entire program. Therefore programs can be more readable and better documented. RL51 can be invoked either manually or through a batch file for improved productivity.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the MCS-51 family. The listing file shows the results of the link/locate process.

LIB51 LIBRARIAN

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects—thereby maintaining the customer's software investment.

ORDERING INFORMATION

Order Code	Operating Environment
D86PLM51	PL/M51 Software for PC DOS 3.0 Systems
iMDX352	PL/M51 Software for Intel 8-bit Development Systems (iPDS, Series II)
I86PLM51	PL/M51 Software for Intel 16-bit Development Systems (SERIES III, Series IV)

Documentation Package

PL/M 51 User's Guide

MCS-51 Utilities User's Guide

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and monthly Technical Newsletters are available.

8051 SOFTWARE DEVELOPMENT PACKAGE

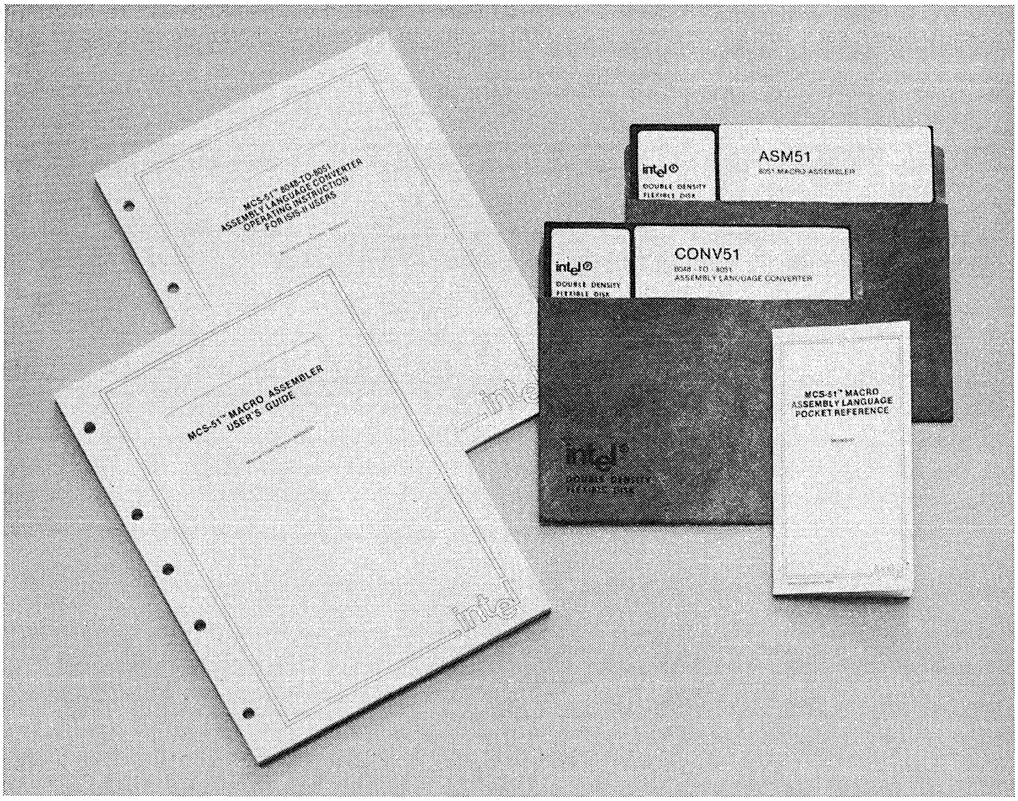
- **Symbolic relocatable assembly language programming for 8051 microcontrollers**
- **Extends Intellec® Microcomputer Development System to support 8051 program development**
- **Produces Relocatable Object Code which is linkable to other 8051 Object Modules**
- **Encourage modular program design for maintainability and reliability**
- **Macro Assembler features conditional assembly and macro capabilities**
- **CONV51 Converter for translation of 8048 assembly language source code to 8051 assembly language source code**
- **Provides upward compatibility from the MCS-48™ family of single-chip microcontrollers**
- **Supports all members of the Intel MCS® 51 architecture**

The 8051 software development package provides development system support for the powerful 8051 family of single chip microcomputers. The package contains a symbolic macro assembler and MCS-48 source code converter.

The assembler produces relocatable object modules from 8051 macro assembly language instructions. The object code modules can be linked and located to absolute memory locations. This absolute object code may be used to program the 8751 EPROM version of the chip. The assembler output may also be debugged using the new family of ICE 5100 emulators or with the ICE-51™ in-circuit emulator.

The converter translates 8048 assembly language instructions into 8051 source instructions to provide software compatibility between the two families of microcontrollers.

Software available for PC DOS 3.0 based IBM* PC XT/AT Systems, iPDS™ Systems, Series II, Series III and Series IV Intel Development Systems.


MCS-51™ 8048-TO-8051 ASSEMBLY LANGUAGE CONVERTER OPERATING INSTRUCTION FOR ISIS-II USERS
MCS-51™ MACRO ASSEMBLER USER'S GUIDE
ASM51
8051 MACRO ASSEMBLER
CONV51
8048-TO-8051 ASSEMBLY LANGUAGE CONVERTER
intel DOUBLE DENSITY FLEXIBLE DISK
MCS-51™ MACRO ASSEMBLY LANGUAGE POCKET REFERENCE

162771-3

8051 MACRO ASSEMBLER

- **Supports 8051 family program development on Intellec® Microcomputer Development Systems**
- **Gives symbolic access to powerful 8051 hardware features**
- **Produces object file, listing file and error diagnostics**
- **Object files are linkable and locatable**
- **Provides software support for many addressing and data allocation capabilities**
- **Symbolic Assembler supports symbol table, cross-reference, macro capabilities, and conditional assembly**

The 8051 Macro Assembler (ASM51) translates symbolic 8051 macro assembly language modules into linkable and locatable object code modules. Assembly language mnemonics are easier to program and are more readable than binary or hexadecimal machine instructions. By allowing the programmer to give symbolic names to memory locations rather than absolute addresses, software design and debug are performed more quickly and reliably. Furthermore, since modules are linkable and relocatable, the programmer can do his software in modular fashion. This makes programs easy to understand, maintainable and reliable.

The assembler supports macro definitions and calls. This is a convenient way to program a frequently used code sequence only once. The assembler also provides conditional assembly capabilities.

Cross referencing is provided in the symbol table listing, showing the user the lines in which each symbol was defined and referenced.

ASM51 provides symbolic access to the many useful addressing features of the 8051 architecture. These features include referencing for bit and byte locations, and for providing 4-bit operations for BCD arithmetic. The assembler also provides symbolic access to hardware registers, I/O ports, control bits, and RAM addresses. ASM51 can support all members of the 8051 family.

Math routines are enhanced by the MULtiply and DIVide instructions.

If an 8051 program contains errors, the assembler provides a comprehensive set of error diagnostics, which are included in the assembly listing or on another file. Program testing may be performed by using the iUP Universal Programmer and iUP F87/51 personality module to program the 8751 EPROM version of the chip.

ICE 5100, ICE51 and EMV51 are available for program debugging.

RL51 LINKER AND RELOCATOR PROGRAM

- **Links modules generated by the assembler**
- **Locates the linked object to absolute memory locations**
- **Enables modular programming of software for efficient program development**
- **Modular programs are easy to understand, maintainable and reliable**

The 8051 linker and relocator (RL51) is a utility which enables 8051 programmers to develop software in a modular fashion. The linker resolves all references between modules and the relocator assigns absolute memory locations to all the relocatable segments, combining relocatable partial segments with the same name.

With this utility, software can be developed more quickly because small functional modules are easier to understand, design and test than large programs.

The number of symbols in the software is very large because the assembler symbol limit applies only per module not the entire program. Therefore programs can be more readable and better documented.

Modules can be saved and used on different programs. Therefore the software investment of the customer is maintained.

RL51 produces two files. The absolute object module file can be directly executed by the 8051 family. The listing file shows the results of the link/locate process.

CONV51
8048 TO 8051 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

- **Enables software written for the MCS-48™ family to be upgraded to run on the 8051**
- **Maps each 8048 instruction to a corresponding 8051 instruction**
- **Preserves comments; translates 8048 macro definitions and calls**
- **Provides diagnostic information and warning messages embedded in the output listing**

The 8048 to 8051 Assembly Language Converter is a utility to help users of the MCS-48 family of microcomputers upgrade their designs with the high performance 8051 architecture. By converting 8048 source code to 8051 source code, the software investment developed for the 8048 is maintained when the system is upgraded.

The goal of the converter (CONV51) is to attain functional equivalence with the 8048 code by mapping each 8048 instruction to a corresponding 8051 instruction. In some cases a different instruction is produced because of the enhanced instruction set (e.g., bit CLR instead of ANL).

Although CONV51 tries to attain functional equivalence with each instruction, certain 8048 code sequences cannot be automatically converted. For example, a delay routine which depends on 8048 execution speed would require manual adjustment. A few instructions, in fact, have no 8051 equivalent (such as those involving P4-P7). Finally, there are a few areas of possible intervention such as PSW manipulation and interrupt processing, which at least require the user to confirm proper translation. The converter always warns the user when it cannot guarantee complete conversion.

CONV51 produces two files. The output file contains the ASM51 source program produced from the 8048 instructions. The listing file produces correlated listings of the input and output files, with warning messages in the output file to point out areas that may require users' intervention in the conversion.

NOTE:
CONV51 is not available with DOS hosted versions.

LIB51 LIBRARIAN

The LIB51 utility enables MCS-51 programmers to create and maintain libraries of software object modules. With this utility, the customer can develop standard software modules and place them in libraries, which programs can access through a standard interface. When using object libraries, the linker will call only object modules that are required to satisfy external references.

Consequently, the librarian enables the customer to port and reuse software on different projects—thereby maintaining the customer's software investment.

ORDERING INFORMATION

Order Code	Operating Environment
D86ASM51	8051 Assembler for PCDOS 3.0 Systems
MCI51ASM	8051 Assembler for 8-bit Intel Development Systems (iPDS™ Systems, Series II)
I86ASM51	8051 Assembler for 16-bit Intel Development Systems (SERIES III, Series IV)

Documentation Package:

MCS-51 Macro Assembler User's Guide

MCS-51 Utilities User's Guide for 8080/8085 Based Development System

MCS-51 8048-to-8051 Assembly Language Converter Operating Instructions for ISIS-II Users

SUPPORT:

Hotline Telephone Support, Software Performance Reporting (SPR), Software Updates, Technical Reports, Monthly Newsletter available.

intel®

MCS®-48 DISKETTE-BASED SOFTWARE SUPPORT PACKAGE

- **Extends Intellec® Microcomputer Development Systems to Support MCS®-48 Development**
- **MCS-48 Assembler Provides Conditional Assembly and Macro Capability**
- **Takes Advantage of Powerful ISIS-II File Handling and Storage Capabilities**
- **Provides Assembler Output in Standard Intel Hex Format**

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operation codes, and provides both conditional and macroassembler programming. Output may be loaded either to an ICE-49 module for debugging or into the iUP Universal PROM Programmer for 8748 PROM programming. The MCS-48 assembler operates under the ISIS-II operating system on Intel Development systems.

Table 1. Sample MCS-48 Diskette-Based

```
ISIS-II 8048 MACROASSEMBLER, V1.0                          PAGE 1

LOC   OBJ                      SEQ        SOURCE STATEMENT

                                1   ;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
                                2   ;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
                                3   ;RESULT IN 'ALPHA.' LENGTH OF NUMBER IS 'COUNT' DIGIT
                                4   ;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
                                5   ;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
                                6   ;ODD)
                                7   INIT     MACRO    AUGND,ADDND,CNT
                                8            MOV      R0, #AUGND
                                9   L1:      MOV      R1, #ADDND
                               10            MOV      R2, #CNT
                               11            ENDM
                               12   ;
0001E                          13   ALPHA    EQU      30
0028                           14   BETA     EQU      40
0032                           15   COUNT    EQU      5
0100                           16            ORG      100H
                               17            INIT     ALPHA, BETA, COUNT
0100  B81E                     18+           MOV      R0, #ALPHA
0102  B928                     19+ L1:       MOV      R1, #BETA
0104  BA32                     20+           MOV      R2, #COUNT
0106  97                       21            CLR      C
0107  F0                       22   LP:      MOV      A, @R0
0108  71                       23            ADDC     A, @R1
0109  57                       24            DA       A
010A  A1                       25            MOV      @R0, A
010B  18                       26            INC      R0
010C  19                       27            INC      R1
010D  EA07                     28            DJNZ     R2, LP
                                             END
USER SYMBOLS
ALPHA  0001E    BETA 0028    COUNT 0005    LP 0107
L1     0102

ASSEMBLY COMPLETE, NO ERRORS


ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE, V1.0             PAGE 1

SYMBOL CROSS REFERENCE

 ALPHA  13#   17
 BETA   14#   17
 COUNT  15#   17
 INIT    7#   17
 L1     19#
 LP     22#   28
```

280381-2

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

October 1986
Order Number: 280381-002

FUNCTIONAL DESCRIPTION

The MCS-48 assembler translates symbolic 8048 assembly language instructions into the appropriate machine operations codes. The ability to refer to program addresses with symbolic names eliminates the errors of hand translation and makes it easier to modify programs when adding or deleting instructions. Conditional assembly permits the programmer to specify which portions of the master source document should be included or deleted in variations on a basic system design, such as the code required to handle optional external devices. Macro capability allows the programmer use of a single label to define a routine. The MCS-48 assembler will assemble the code required by the reserved routine whenever the macro label is inserted in the text. Output from the assembler is in standard Intel hex format. It may be either loaded directly to an in-circuit emulator (ICE-49) module for integrated hardware/software debugging, or loaded into the iUP Universal PROM Programmer for 8748 PROM programming. A sample assembly listing is shown in Table 1.

The MCS-48 assembler supports the 8048, 8049, 8050, 8020, 8021, 8022, 8041 and 8042. The MSC-48 assembler can also support CMOS versions of the 8048 family.

SPECIFICATIONS

Operating Environment

(All) Intel Microcomputer Development Systems (Series II, Series III/Series IV)
Intel Personal Development System

Documentation Package

Titles of: User Guides
Operating Instructions
Reference Manuals

Ordering Information

Part Number	Description
MDS-D48	MCS-48 Disk Based Assembler

Requires Software License

SUPPORT

Hotline Telephone Support, Software Performance Reports (SPR), Software Updates, Technical Reports, Monthly Newsletters are available.

MCS®-96 SOFTWARE DEVELOPMENT PACKAGES

- **Choice of Hosts**
- **MCS®-96 Software Support Package**
- **Supports All Members of the MCS®-96 Family**
- **PL/M-96 Software Package**

MCS®-96 SOFTWARE SUPPORT PACKAGE

- **Symbolic relocatable assembly language programming for the 8096 microcontroller family**
- **System Utilities for Program Linking and Relocation**
- **Extends Intellec® Microcomputer Development System to support MCS-96 program development**
- **Encourages modular program design for maintainability and reliability**

The MCS®-96 Software Support Package provides development system support for the MCS-96 family of 16-bit single chip microcomputers. The support package includes a macro assembler and system utilities.

The assembler produces relocatable object modules from MCS-96 macro assembly language instructions. The object modules then are linked and located to absolute memory locations.

The assembler and utilities run on the Intellec® Series III or equivalent Microcomputer Development System and PC DOS 3.0 IBM* PC XT/AT* Systems.

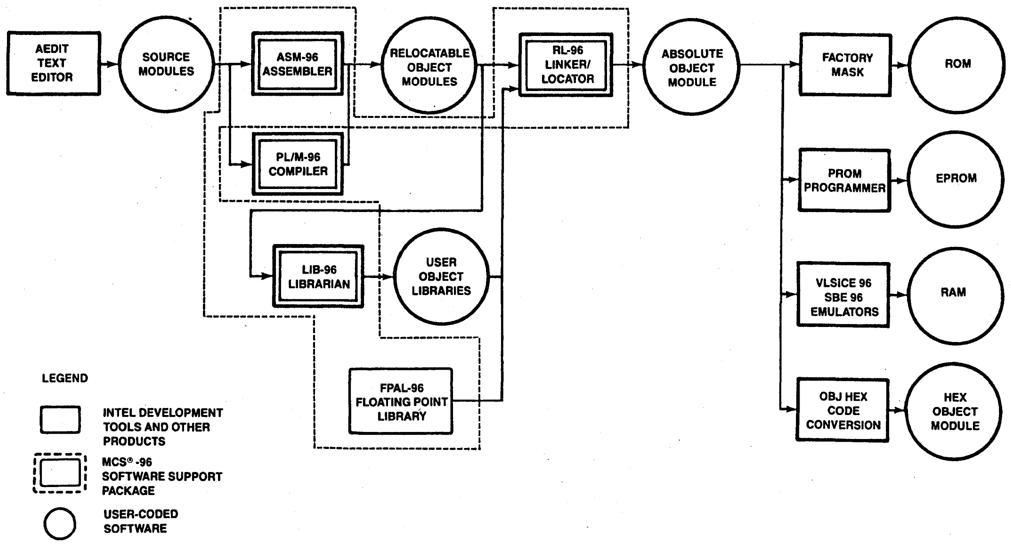


Figure 1. MCS®-96 Software Development Process

*IBM and AT are registered trademarks of International Business Machines Corporation.

October 1986
Order Number: 230613-005

8096 MACRO ASSEMBLER

- **Supports 8096 Family Program Development on Intellec® Microcomputer Development System or IBM PC XT/AT**
- **Gives Symbolic Access to Powerful 8096 Hardware Features**
- **Object Files are Linkable and Locatable**
- **Symbolic Assembler Supports Macro Capabilities, Cross Reference, Symbol Table and Conditional Assembly**

ASM-96 is the macro assembler for the MCS family of microcontrollers. ASM-96 translates symbolic assembly language mnemonics into relocatable object code. Since the object modules are linkable and locatable, ASM-96 encourages modular programming practices.

The macro facility in ASM-96 allows programmers to save development and maintenance time since common code sequences only have to be done once. The assembler also provides conditional assembly capabilities.

ASM-96 supports symbolic access to the many features of the 8096 architecture. An "include" file is provided with all of the 8096 hardware registers defined. Alternatively, the user can define any subset of the 8096 hardware register set.

Math routines are supported with mnemonics for 16 × 16-bit multiply or 32/16-bit divide instructions.

The assembler runs on a Series III/Series IV Intellec Development System or on a PC-DOS 3.0 IBM PC XT/AT.

RL96 LINKER AND RELOCATOR PROGRAM

- **Links Modules Generated by ASM-96 and PL/M-96**
- **Locates the Linked Object Module to Absolute Memory Locations**
- **Encourages Modular Programming for Faster Program Development**
- **Automated Selection of Required Modules from Libraries to Satisfy Symbolic References**

RL96 is a utility that performs two functions useful in MCS-96 software development:

— The link function which combines a number of MCS-96 object modules into a single program.

— The locate functions which assigns an absolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:

— The program or absolute object module file that can be executed by the targeted member of the MCS-96 family.

— The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocator allows programmers to concentrate on software functionally and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- **Implements IEEE Floating Point Arithmetic**
- **Basic Arithmetic Operations +, −, ×, /, Mod Plus Square Root**
- **Supports Single Precision 32 Bit Floating Point Variables**
- **Includes an Error Handler Library**

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

—CREATE:	Creates an empty library file.
—ADD:	Adds object modules to a library file.
—DELETE:	Deletes object modules from a library file.
—LIST:	Lists the modules in the library file.
—EXIT:	Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

ORDERING INFORMATION

Order Code	Operating Environment
D86ASM96	96 Assembler for PC DOS 3.0 Systems
I86ASM96	96 Assembler for Intel Development Systems (Series III and Series IV)

Documentation Package:

MCS-96 Macro Assembler User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket Reference Card
8096 Floating Point Arithmetic Library

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

PL/M-96 SOFTWARE PACKAGE

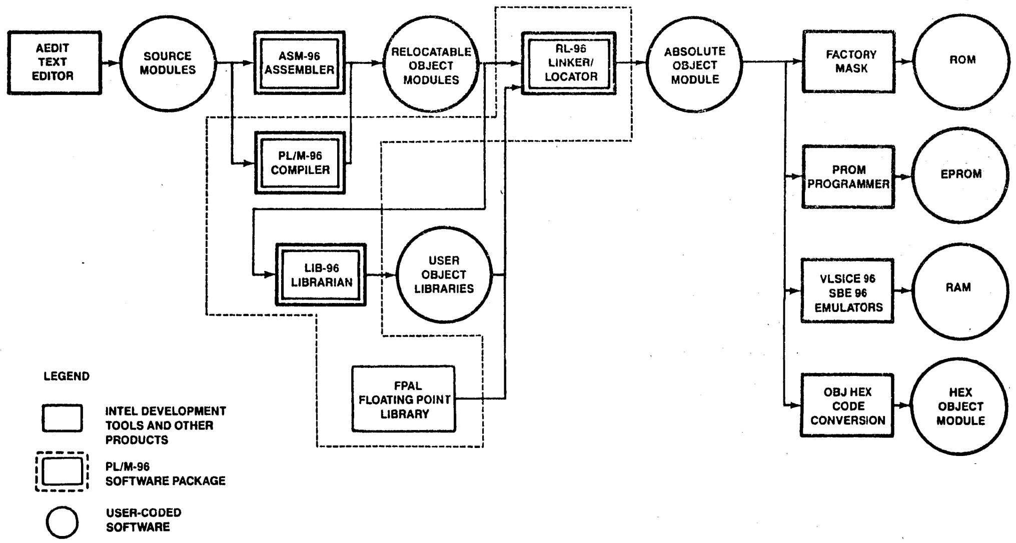
- **Choice of Hosts**
- **Block Structured Language Design Encourages Module Programming**
- **Provides Access to MCS®-96 on Chip Resources**
- **Produces Relocatable Object Code which is Linkable to Object Modules Generated by Other MCS®-96 Translators**
- **Resident on iAPX-86 Intel Microcomputer Development Systems for Higher Performance**
- **Includes a Linking and Relocating Utility and the Library Manager**
- **IEEE Floating Point Library included for Numeric Support**
- **Compatible with PL/M-86 Assuring Design Portability**

PL/M-96 is a structured, high-level programming language useful for developing software for the Intel MCS-96 family of microcontrollers. PL/M-96 was designed to support the software requirements of advanced 16 bit microcontrollers. Access to the on chip resources of the MCS-96 has been provided in PL/M-96.

PL/M-96 is compatible with PL/M-86. Programmers familiar with PL/M will find they can program in PL/M-96 with little relearning effort.

The PL/M-96 compiler translates PL/M-96 high level language statements into MCS-96 machine instructions. By programming in PL/M an engineer can be more productive in the initial software development cycle of the project. PL/M can also reduce future maintenance and support cost because PL/M programs are easier to understand. PL/M-96 was designed to complement Intel's ASM-96.

PL/M-96 is available for Intel Series III and Series IV Development Systems and for PC DOS 3.0 based IBM* PC XT/AT* Systems.



230613-2

Figure 2. PL/M-96 Software Package

PL/M-96 COMPILER

FEATURES

Major features of the PL/M-96 compiler and programming language include:

Structured Programming

Programs written in PL/M-96 are developed as a collection of procedures, modules and blocks. Structured programs are easier to understand, maintain and debug. PL/M-96 programs can be made more reliable by clearly defining the scope of user variables (for example, local variables in a procedure). REENTRANT procedures are also supported by PL/M-96.

Language Compatibility

PL/M-96 object modules are compatible with all other object modules generated by Intel MCS-96 translators. Programmers may choose to link ASM-96 and PL/M-96 object modules together.

PL/M-96 object modules were designed to work with other Intel support tools for the MCS-96. The DEBUG compiler control provides these tools with symbolic information.

Data Types Supported

PL/M-96 supports seven data types for programmer flexibility in various logical, arithmetic and addressing functions. The seven data types include:

—BYTE:	8-bit unsigned number
—WORD:	16-bit unsigned number
—DWORD:	32-bit unsigned number
—SHORTINT:	8-bit signed number
—INTEGER:	16-bit signed number
—LONGINT:	32-bit signed number
—REAL:	32-bit floating point number

Another powerful feature are BASED variables. BASED variables allow the user to map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Data Structures Supported

Two data structuring facilities are supported by PL/M-96. The user can organize data into logical groups. This adds flexibility in referencing data.

— Array: Indexed list of same type data elements

— Structure: Named collection of same or different type data elements

— Combinations of Both: Arrays of structures or structures of arrays

Interrupt Handling

Interrupts are supported in PL/M-96 by defining a procedure with the INTERRUPT attribute. The compiler will generate code to save and restore the program status word when handling hardware interrupts of the MCS-96.

Compiler Controls

Compile time options increase the flexibility of the PL/M-96 compiler. These controls include:

— Optimization

— Conditional compilation

— The inclusion of common PL/M-96 source files from disk

— Cross reference of symbols

— Optional assembly language code in the listing file

Code Optimizations

The PL/M-96 compilers has four levels of optimization for reducing program size.

- Combination of constant expressions; "Strength reductions" (e.g.: a shift left rather than multiply by two)
- Machine code optimizations; elimination of superfluous branches; reuse of duplicate code, removal of unreachable code
- Overlaying of on chip RAM variables
- Optimization of based variable operations
- Use of short jumps where possible

Built in Functions

An extensive list of built in functions has been supplied as part of the PL/M-96 language. Besides TYPE CONVERSION functions, there are built in functions for STRING manipulations. Functions are provided for interrogating the MCS-96 hardware flags such as CARRY and OVERFLOW.

Error Checking

If the PL/M-96 compiler detects a programming or compilation error, a fully detailed error message is provided by the compiler. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This powerful PL/M-96 feature can yield a two times increase in throughput when a user is in the initial program development cycle.

BENEFITS

PLM-96 is designed to be an efficient, cost-effective solution to the special requirements of MCS-96 Microcontroller Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M-96 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M-96, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M-96 is designed to aid in the development of reliable software (PL/M programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status. The more simply the program is stated, the more likely it is to perform its intended function.

Easier Enhancements and Maintainance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

RL96 LINKER AND RELOCATOR PROGRAM

- **Links Modules Generated by ASM-96 and PL/M-96**
- **Locates the Linked Object Module to Absolute Memory Locations**
- **Encourages Modular Programming for Faster Program Development**
- **Automated Selection of Required Modules from Libraries to Satisfy Symbolic References**

RL96 is a utility that performs two functions useful in MCS software development:
— The link function which combines a number of MCS object modules into a single program.
— The locate function which assigns an obsolute address to all relocatable addresses in the MCS-96 object module.

RL96 resolves all external symbol references between modules and will select object modules from library files if necessary.

RL96 creates two files:
— The program or absolute object module file that can be executed by the targeted member of the MCS family.
— The listing file that shows the results of link/locate, including a memory map symbol table and an optional cross reference listing.

The relocator allows programmers to concentrate on software functionality and not worry about the absolute addresses of the object code. RL96 promotes modular programming. The application can be broken down into separate modules that are easier to design, test and maintain. Standard modules can be developed and used in different applications thus saving software development time.

FPAL96 FLOATING POINT ARITHMETIC LIBRARY

- **Implements IEEE Floating Point Arithmetic**
- **Basic Arithmetic Operations +, −, ×, /, Mod Plus Square Root**
- **Supports Single Precision 32 Bit Floating Point Variables**
- **Includes an Error Handler Library**

FPAL96 is a library of single precision 32-bit floating point arithmetic functions. All math adheres to the proposed IEEE floating point standard for accuracy and reliability. An error handler to handle exceptions (for example, divide by zero) is included.

The following functions are included:

ADD	NEGATE
SUBTRACT	ABSOLUTE
MULTIPLY	SQUARE ROOT
DIVIDE	INTEGER
COMPARE	REMAINDER

LIB 96

The LIB 96 utility creates and maintains libraries of software object modules. The customer can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces.

LIB 96 uses the following set of commands:

—CREATE:	Creates an empty library file
—ADD:	Adds object modules to a library file
—DELETE:	Deletes object modules from a library file
—LIST:	Lists the modules in the library file
—EXIT:	Terminates LIB 96

When using object libraries, RL96 will include only those object modules that are required to satisfy external references, thus saving memory space.

ORDERING INFORMATION

Order Code	Operating Environment
D86PLM96	PL/M-96 Compiler for PC DOS 3.0 based Systems
I86PLM96	PL/M-96 Compiler for Intel Series III and Series IV Development Systems

Documentation Package:

PL/M-96 User's Guide
MCS-96 Utilities User's Guide
MCS-96 Assembler and Utilities Pocket Reference Card
8096 Floating Point Arithmetic Library

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available.

VAX*/VMS* RESIDENT SOFTWARE DEVELOPMENT PACKAGES FOR 80286

- **Hosted on DEC VAX*/MicroVAX Minicomputers Under the VMS* Operating System**
- **Allows Development of System and Application Software for the Protected Virtual Address Mode of the 80286**
- **Packages include PL/M-286, BUILD-286, BIND-286, LIB-286 and MAP-286**
- **Compatible with Corresponding Intel Development System Resident Products**

These packages provide the capability of developing software on a VAX*/VMS* host for the 80286 in protected virtual address mode. With these packages a user can assemble and compile 286 programs, configure system and application software and create and manage 286 object libraries. Figure 1 illustrates the process of 286 software development on VAX*/VMS* hosts.

Two packages are available:

1. A PL/M-286 package which contains the PL/M-286 compiler and run time support libraries.
2. An ASM-286 package which contains the 80286 Assembler (ASM-286) and programming utilities. These utilities include the 80286 System Builder (BLD-286), the System Binder (BND-286), a Library Utility (LIB-286) and an Object Map Utility (MAP-286).

These packages are compatible with corresponding products which are hosted on Intel development systems. Correspondence can be established via version numbers. For example, BND-286 V2.0 offers the same set of features on VAX/VMS and Intel development systems.

Owing to this compatibility, 80286 software developed on VAX/VMS can be linked to 80286 software from development systems. Moreover, 80286 programs developed on the VAX can then be downloaded to development systems and debugged using 286 debuggers like the I2ICE™-286 system.

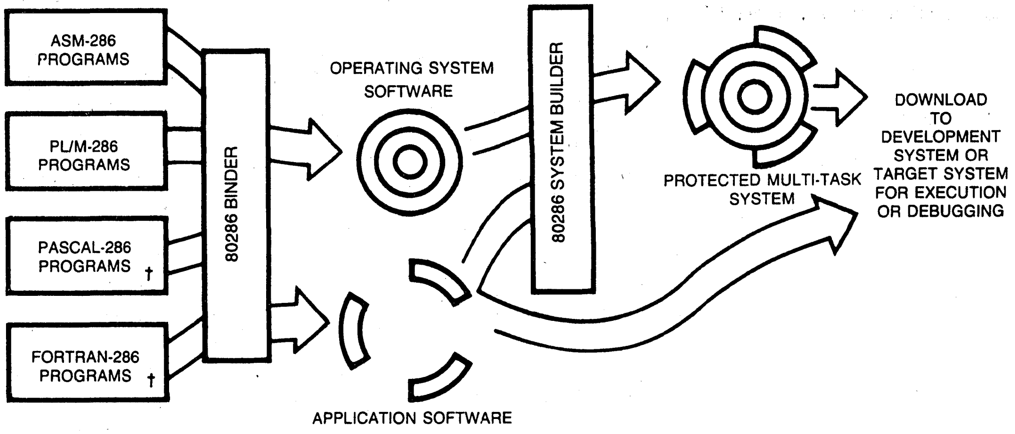


Figure 1. 286 Software Development on VAX*/VMS*

*VAX, VMS are trademarks of Digital Equipment Corporation

†Currently Available on Intel Development Systems Only

October 1986
Order Number: 231038-002

VAX*/VMS* RESIDENT PL/M-286

- **Hosted on DEC VAX*/MicroVAX Minicomputers Under the VMS* Operating System**
- **Systems Programming Language for the Protected Virtual Address Mode 80286**
- **Enhanced to Support Design of Protected, Multi-User, Multi-Tasking, Virtual Memory Operating System Software**
- **Provides Multiple Levels of Optimization to Produce Efficient Code**
- **Produces Relocatable Object Code Linkable to Object Modules Generated by Other Intel 286 Language Translators**
- **Upward Compatible with PL/M-86 and PL/M-80 to Allow Software Portability**
- **Compatible with Development System Resident PL/M-286**

PL/M-286 is a powerful, structured, high-level system implementation language for the development of system software for the protected virtual address mode 80286. PL/M-286 has been enhanced to utilize 80286 features-memory management and protection–for the implementation of multi-user, multi-tasking virtual memory operating systems.

PL/M-286 is upward compatible with PL/M-86 and PL/M-80. Existing systems software can be re-compiled with PL/M-286 to execute in protected virtual address mode on the 80286.

PL/M-286 is the high-level alternative to assembly language programming on the 80286. For the majority of 80286 system programs, PL/M-286 provides the features needed to access and to control efficiently the underlying 80286 hardware, and consequently it is the cost-effective approach to develop reliable, maintainable system software.

The PL/M-286 compiler has been designed to efficiently support all phases of software development. Features such as built-in syntax checker, multiple levels of optimization, virtual symbol table and four models of program size and memory usage for efficient code generation provide the total program development support needed. The compiler also provides complete symbolic debug capability to the various 286 debuggers and emulators.

VAX/VMS resident PL/M-286 is completely feature compatible with development system resident PL/M-286 with the same version number.

VAX*/VMS* RESIDENT 80286 MACRO ASSEMBLER

- **Supports Full Instruction Set of the 80286 including Memory Protection and Numerics (with 80287)**
- **Structures and RECORDS Provide Powerful Data Representation**
- **Type Checking at Assembly Time Helps Reduce Errors at Run-Time**
- **Powerful and Flexible Text Macro Facility**
- **Upward Compatible with ASM-86/88/186**
- **Compatible with Development System Resident 80286 Macro Assembler**

ASM-286 is the "high-level" macro assembler for the 80286 assembly language. ASM-286 translates symbolic assembly language mnemonics into relocatable object code. The assembler mnemonics are a superset of ASM-86/88 mnemonics; new ones have also been added to support the new 80286 instructions. The segmentation directives have been greatly simplified.

The 80286 assembly language includes approximately 150 instruction mnemonics. From these few mnemonics the assembler can generate over 4,000 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 150 mnemonics to generate all possible machine instructions. ASM-286 generates the shortest machine instruction possible (given explicit information as to the characteristics of any forward referenced symbols).

The powerful macro facility in ASM-286 saves development and maintenance time by coding common program sequences only once. A macro substitution is made each time the sequence is to be used. This facility also allows for conditional assembly of certain program sequences.

ASM-286 offers many features normally found only in high-level languages. The assembly language is strongly typed, which means it performs extensive checks on the usage of variables and labels. This means that many programming errors will be detected when the program is assembled, long before it is being debugged.

ASM-286 object modules conform to a thorough, well-defined format used by 286 high-level languages and utilities. This makes it easy to call (and be called from) HLL object modules.

ASM-286 also provides support for the 80287 numerics co-processor. The complete instruction set of the 80287 is available through high-level mnemonics.

VAX/VMS resident ASM-286 is completely feature compatible with development system resident ASM-286 with the same version number.

VAX*/VMS* RESIDENT 80286 SYSTEM BUILDER

- **A Tool for Configuring Multi-Tasking Protected, Virtual Memory Systems Software for the 80286**
- **Links Separately Compiled Modules Resolves EXTERNAL/PUBLIC Definitions**
- **Creates a Memory Image of a 286 System for Cold Start Execution**
- **Target System May Be Bootloadable, Programmed into ROM or Loaded from Mass Storage**
- **Generates Print File with Command Listing and System Map**
- **Compatible with Development System Resident 80286 System Builder**

BLD-286 is the 80286 System Builder. It allows systems programmers to configure multi-tasking and memory protected 80286 software. The configuration is specified by the user in a "Build file" using a symbolic meta-language. BLD-286 thus provides the programmer a high-level symbolic interface to the multi-tasking and memory protection features of the 80286 architecture.

BLD-286 accepts as inputs object modules from the 80286 translators, the 80286 Binder and itself (for incremental building). Using the programmer's specifications in the Build File, it produces a bootloadable or loadable module as well as a print file with a map of the configured module.

Using the builders command language, system programmers may perform the following functions:
— Assign physical addresses to segments; also set segment access rights and limits.
— Create Call, Trap, and Interrupt "Gates" (entry-points) for inter-level program transfers.
— Make gates available to tasks; this is an easier way to define program interfaces than using interface libraries.
— Create Global (GDT), Interrupt (IDT), and any Local (LDT) Descriptor Tables.
— Create Task State Segments and Task Gates for multi-tasking applications.
— Resolve inter-module and inter-level references, and perform type-checking.
— Automatically select required modules from libraries.
— Configure the memory image into partitions in the address space.
— Selectively generate an object file and various sections of the print file.

VAX/VMS BLD-286 is completely feature compatible with development system resident BLD-286 with the same version number.

VAX*/VMS* RESIDENT 80286 BINDER

- **Links Separately Compiled Program Modules into an Executable Task**
- **Makes the 80286 Protection Mechanism Invisible to Application Programmers**
- **Assigns Virtual Addresses to Tasks**
- **Performs Incremental Linking with Output of Binder and Builder**
- **Resolves PUBLIC/EXTERNAL Code and Data References, and Performs Intermodule Type-Checking**
- **Provides Print File Showing Segment Map, Errors and Warnings**
- **Generates Linkable or Loadable Module for Debugging**
- **Compatible with Development System Resident 80286 Binder**

BND-286 is a utility that combines 80286 object modules into executable tasks. In creating a task, the Binder resolves Public and External symbol references, combines segments, and performs address fix-ups on symbolic code and data.

The Binder takes object modules, produced by the 286 translators, and generates a loadable module (for execution or debugging), or a linkable module (to be re-input to the Binder later; this is called incremental binding). The binder accepts library modules as well, linking only those modules required to resolve external references. BND-286 generates a print file displaying a segment map, and error messages.

The Binder is useful for system as well as application programmers. Since application programmers need to develop software independent of any system architecture, the 286 memory protection mechanism is "hidden" from users of the Binder. This allows application tasks to be fully debugged before becoming part of a protected system. (A protected system may be debugged, as well.) System protection features are specified later in the development cycle, using the 286 System Builder. It is possible to link operating system services required by a task using either the Binder or the Builder. This flexibility adds to the ease of use of the 286 utilities.

VAX/VMS resident BND-286 is completely feature compatible with development system resident BND-286 with the same version number.

VAX*/VMS* RESIDENT 80286 LIBRARIAN

- **Allows Creation and Management of 80286 Object Libraries**
- **Library Functions include Create, Delete, Add, Replace, Copy, Save, Backup and Display**
- **Only Required Modules Linked in When Using Binder or Builder**
- **Compatible with Development System Resident 80286 Librarian**

LIB-286 is the 80286 Librarian. It can be used to create and manage 80286 Object Libraries. By placing often used object modules into libraries, the administrative overhead of managing software modules can be reduced.

VAX/VMS based LIB-286 is completely feature compatible with development system resident LIB-286 with the same version number.

VAX*/VMS* RESIDENT 80286 MAPPER

- **Flexible Utility to Display Object File Information in Symbolic Form**
- **Compatible with Development System Resident 80286 Mapper**

MAP-286 is a cross reference utility for 80286 object modules. It provides a symbolic listing of the EXTERNAL and PUBLIC symbols in the specified object modules.

VAX/VMS resident MAP-286 is completely feature compatible with development system resident MAP-286 with the same version number.

SPECIFICATIONS

Operating Environment

DEC VAX* 11/780 or compatible model running VMS* operating system V3.4 (or upward compatible versions)

Documentation

Installation guide and user's manuals for the software are supplied with the products.

*VAX/VMS are trademarks of Digital Equipment Corporation

SUPPORT

Hotline Telephone Support, Software, Performance Report (SPR) Software Updates, Technical Reports and Monthly Newsletters are available.

ORDERING INFORMATION

Product Code	Description
iMDX-371VX	ASM-286, BLD-286, BND-286, LIB-286, MAP-286
iMDX-373VX	PL/M-286

VAX*/VMS* RESIDENT 8086/88/186 SOFTWARE DEVELOPMENT PACKAGES

- Executes on DEC VAX* Minicomputer under VMS* Operating System to translate PL/M-86, Pascal-86 and ASM-86 Programs for 8086, 88 and 186 Microprocessors.
- Packages include Pascal-86; PL/M-86; ASM-86; Link and Relocation Utilities; OH-86 Absolute Object Module to Hexadecimal Format Converter; and Library Manager Program.
- Output linkable with Code Generated on Intellec® Development Systems.

The VAX/VMS Resident Software Development Packages contain software development tools for the 8086, 88, and 186 microprocessors. The package lets the user develop, compile, maintain libraries, and link and locate programs on a VAX running the VMS operating system. The translator output is object module compatible with programs translated by the corresponding version of the translator on an Intellec Development System.

Four packages are available:

1. An ASM-86 Assembler Package which includes the Assembler, the Link Utility, the Locate Utility, the absolute object to hexadecimal format conversion utility and the Library Manager Program.
2. A PL/M-86 Compiler Package which contains the PL/M-86 Compiler and Runtime Support Libraries.
3. A Pascal-86 Compiler Package which contains the Pascal-86 Compiler and Runtime Support Libraries.
4. A C-86 Compiler Package which contains the C-86 Compiler and Run-Time Libraries.

The VAX/VMS resident development packages and the Intellec Development System development packages are built from the same technology base. Therefore, the VAX/VMS resident development packages and the Intellec Development System development packages are very similar.

Version numbers can be used to identify features correspondence. The VAX/VMS resident development packages will have the same features as the Intellec Development System product with the same version number.

Support for the 80186 processor will be provided as an update to the 8086, 88 software.

The object modules produced by the translators contain symbol and type information for programming debugging using ICE™ translators and/or the PSCOPE debugger. For final production version, the compiler can remove this extra information and code.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

October 1986
Order Number: 210643-003

VAX*-PL/M-86/88/186 SOFTWARE PACKAGE

- **Executes on VAX*/MICROVAX Minicomputers under the VMS* Operating System**
- **Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard**
- **Easy-To-Learn Block-Structured Language Encourages Program Modularity**
- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX*, a PC XT/AT running PC-DOS Version 3.0 or Intellec® Development Systems**
- **Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization**
- **Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors**
- **Source Input/Object Output Compatible with PL/M-86 Hosted on an Intellec® Development System**
- **ICE™, PSCOPE Symbolic Debugging Fully Supported**

Like its counterpart for MCS®-80/85 program development, and Intellec® hosted 8086 program development, VAX-PL/M-86 is an advanced, structured high-level programming language. The VAX-PL/M-86 compiler was created specifically for performing software development for the Intel 8086, 88 and 186 Microprocessors.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The VAX-PL/M-86 compiler efficiently converts free-form PL/M language statements into equivalent 8086/88/186 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-PASCAL-86/88 SOFTWARE PACKAGE

- **Executes VAX*/MICROVAX Minicomputers under the VMS* Operating System**
- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086 Object Modules, Generated on Either a VAX*, a PC XT/AT running PC-DOS Version 3.0 or Intellec® Development Systems**
- **ICE™, PSCOPE Symbolic Debugging Fully Supported**
- **Implements REALMATH for Consistent and Reliable Results**
- **Supports 8086/20, 88/20 Numeric Data Processors**
- **Strict Implementation of ISO Standard Pascal**
- **Useful Extensions Essential for Microcomputer Applications**
- **Separate Compilation with Type-Checking Enforced between Pascal Modules**
- **Compiler Option to Support Full Run-Time Range-Checking**
- **Source Input/Object Output Compatible with Pascal-86 Hosted on a Intellec® Development System**

VAX-PASCAL-86 conforms to and implements the ISO Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. Other extensions include additional data types not required by the standard and miscellaneous enhancements such as an allowed underscore in names, an OTHERWISE clause in CASE construction and so forth. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The VAX-PASCAL-86 compiler runs on the Digital Equipment Corporation VAX under the VMS Operating System. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs on the target system as an alternate to the development system environment. Program modules compiled under PASCAL-86 are compatible and linkable with modules written in PL/M-86, and ASM-86. With a complete family of compatible programming languages for the 8086, 88, and 186 one can implement each module in the language most appropriate to the task at hand.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX* 8086/88/186 MACRO ASSEMBLER

- **Executes on VAX*/MICROVAX Minicomputers under The VMS* Operating System**
- **Produces Relocatable Object Code Which is Linkable to All Other Intel 8086/88/186 Object Modules, Generated on Either a VAX*, a PC XT/AT running PC-DOS Version 3.0 or Intellec® Development Systems**
- **Powerful and Flexible Text Macro Facility with Three Macro Listing Options to Aid Debugging**
- **Highly Mnemonic and Compact Language, Most Mnemonics Represent Several Distinct Machine Instructions**
- **"Strongly Typed" Assembler Helps Detect Errors at Assembly Time**
- **High-Level Data Structuring Facilities Such as "STRUCTURES" and "RECORDS"**
- **Over 120 Detailed and Fully Documented Error Messages**
- **Produces Relocatable and Linkable Object Code**
- **Source Input/Object Output Compatible with ASM-86 hosted on an Intellec® Development System**

VAX-ASM-86 is the "high-level" macro assembler for the 8086/88/186 assembly language. VAX-ASM-86 translates symbolic 8086/88/186 assembly language mnemonics into 8086/88/186 relocatable object code.

VAX-ASM-86 should be used where maximum code efficiency and hardware control is needed. The 8086/88/186 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 8086/88/186 machine instructions. VAX-ASM-86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

VAX-ASM-86 offers many features normally found only in high-level languages. The 8086/88/186 assembly language is strongly typed. The assembler performs extensive checks on the usage of variable and labels. The assembler uses the attributes which are derived explicity when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be deteced when the program is assembled, long before it is being debugged on hardware.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-LIB-86

- **Executes on VAX*/MICROVAX Minicomputers under the VMS* Operating System**
- **VAX-LIB-86 is a Library Manager Program which Allows You to:**
  Create Specifically Formatted Files to Contain Libraries of Object Modules
  Maintain These Libraries by Adding or Deleting Modules
  Print a Listing of the Modules and Public Symbols in a Library File
- **Libraries Can be Used as Input to VAX-LINK-86 Which Will Automatically Link Modules from the Library that Satisfy External References in the Modules Being Linked**
- **Abbreviated Control Syntax**

Libraries aid in the job of building programs. The library manager program VAX-LIB-86 creates and maintains files containing object modules. The operation of VAX-LIB-86 is controlled by commands to indicate which operation VAX-LIB-86 is to perform. The commands are:

CREATE:	creates an empty library file
ADD:	adds object modules to a library file
DELETE:	deletes modules from a library file
LIST:	lists the module directory of library files
EXIT:	terminates the LIB-86 program and returns control to VMS

When using object libraries, the linker will call only those object modules that are required to satisfy external references, thus saving memory space.

VAX-OH-86

- **Executes on VAX*/MICROVAX Minicomputers under the VMS* Operating System**
- **Converts an 8086/88/186 Absolute Object Module to Symbolic Hexademical Format**
- **Facilitates Preparing a file for Loading by Symbolic Hexadecimal Loader (e.g. iSBC® Monitor SDK-86 Loader), or Universal PROM Mapper**
- **Converts an Absolute Module to a More Readable Format that can be Displayed on a CRT or Printed for Debugging**

The VAX-OH-86 utility converts an 86/88 absolute object module to the hexadecimal format. This conversion may be necessary for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or the Universal PROM Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute form; the output from VAX-LOC-86 is in absolute format.

*VAX, VMS are trademarks of Digital Equipment Corporation.

VAX*-LINK-86

- **Executes on VAX*/MICROVAX Minicomputers under the VMS* Operating System**
- **Automatic Combination of Separately Compiled or Assembled 86/88/186 Programs into a Relocatable Module, Generated on Either a VAX, a PC XT/AT running PC-DOS Version 3.0 or an Intellec® Development System**
- **Automatic Selection of Required Modules from Specified Libraries to Satisfy Symbolic References**
- **Extensive Debug Symbol Manipulation, allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively**
- **Automatic Generation of a Summary Map Giving Results of the LINK-86 Process**
- **Abbreviated Control Syntax**
- **Relocatable modules may be Merged into a Single Module Suitable for Inclusion in a Library**
- **Supports "Incremental" Linking**
- **Supports Type Checking of Public and External Symbols**

VAX-LINK-86 combines object modules specified in the VAX-LINK-86 input list into a single output module. VAX-LINK-86 combines segments from the input modules according to the order in which the modules are listed.

VAX-LINK-86 will accept libraries and object modules built from VAX-PL/M-86, VAX-PASCAL-86, VAX-ASM-86, or any other Intel translator generating 8086 Relocatable Object Modules, such as the Series III resident translators.

Support for incremental linking is provided since an output module produced by VAX-LINK-86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

VAX-LINK-86 supports type checking of PUBLIC and EXTERNAL symbols reporting a warning if their types are not consistent.

VAX-LINK-86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the VAX-LINK-86 process and to control the content of the output module.

VAX-LINK-86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using VAX-LOC-86 and enter final testing with much of the work accomplished.

*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

VAX*-LOC-86

- Executes on the VAX*/MICROVAX Minicomputers under the VMS* Operating System
- Automatic Generation of a Summary Map Giving Starting Address, Segment Addresses and Length, and Debug Symbols and their Addresses
- Extensive Capability to Manipulate the Order and Placement of Segments in 8086/8088 Memory
- Abbreviated Control Syntax
- Automatic and Independent Relocation of Independent Relocation of Segments. Segments May be Relocated to Best Match Users Memory Configuration
- Extensive Debug Symbol Manipulation, Allowing Line Numbers, Local Symbols, and Public Symbols to be Purged and Listed Selectively

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

VAX-LOC-86 converts relative addresses in an input module in iAPX-86/88/186 object module format to absolute addresses. VAX-LOC-86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

VAX-LOC-86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the VAX-LOC-86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program and then simply relocate the object code to suit your application.

SPECIFICATIONS

Operating Environment

Required Hardware

VAX* 11/780, 11/782, 11/750, or 11/730 9 Track Magnetic Tape Drive, 1600 BPI

Required Software

VMS Operating System V3.0 or Later. All of the development packages are delivered as unlinked VAX object code which can be linked to VMS as designed for the system where the development package is to be used. VMS command files to perform the link are provided.

Documentation Package

iAPX-86, 88 Development Software Installation Manual and User's Guide for VAX/VMS, Order number 121950-001

Shipping Media

9 Track Magnetic Tape 1600 bpi

ORDERING INFORMATION

Part Number	Description
iMDC-341VX	VAX-ASM-86, VAX-LINK-86, VAX-LOC-86, VAX-LIB-86, VAX-OH-86, Package
iMDX-343VX	VAX-PLM-86 Package
iMDX-344VX	VAX-PASCAL-86 Package

REQUIRES SOFTWARE LICENSE

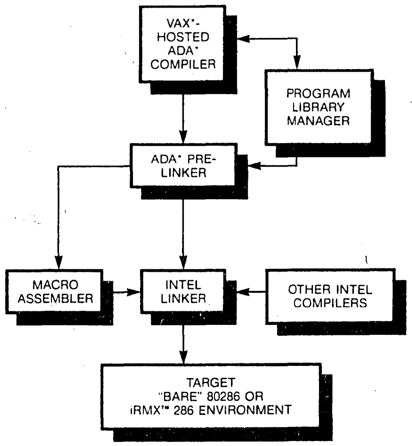
*VAX, DEC, and VMS are trademarks of Digital Equipment Corporation.

intel PRODUCT BRIEF

Ada* 286 Compilation System

The Ada* 286 compilation system is a full, production-quality implementation of the Ada language designed to generate compact, high-quality code for embedded 80286 applications. The compiler will be validated in accordance with U.S. DoD validation requirements, to insure a correct implementation. Two separate runtime environments are provided, giving the flexibility of developing for either bare 80286 or iRMX™ 286 target systems. The Ada 286 development support features full compatibility with existing Intel linking utilities, languages, debuggers and networking, as well as the 80287 coprocessor.

Intel Ada* 286 Compilation System



Product Highlights

- —**Ada* features directly supported by 80286 hardware wherever possible**
- —**Protected mode bare 80286 and iRMX™ 286 targets available**
- —**Highly optimized for efficient 80286/80287 code generation**
- —**Configurable runtime environments can be customized to the application**
- —**Linkable with other Intel languages**
- —**Completely compatible with existing Intel high-level debuggers (I²ICE and PSCOPE)**
- —**VAX*/VMS hosted**

Product Description

Ada 286 is a highly integrated compilation system designed to provide both highly optimized target code and an efficient software development environment.

The compilation system consists of an Ada 286 compiler, program library manager, pre-linker, and 2 separate target environments. All of the Ada 286 components use the common 80286 Object Module Format (OMF) to permit linking of Ada object modules with modules from other translators, as well as on-line code debugging.

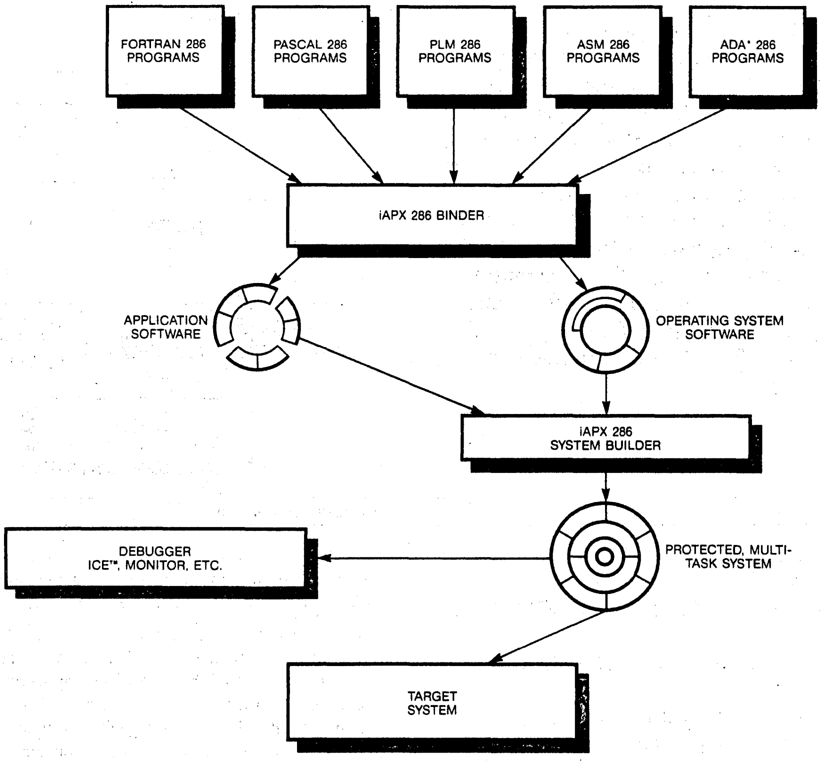
Additionally, the Ada 286 compiler and runtime environments map Ada language constructs onto 80286 hardware features wherever possible, thus giving the performance benefits that a direct hardware implementation allows. This allows complex software systems to be built taking advantage of Ada constructs such as built-in tasking and interrupt handling with optimal performance.

Support is provided for real-time embedded systems development by providing the runtime environments, one for a bare 80286 (no target operating system) and one for an iRMX286 target environment, which are modularly constructed to be able to be reconfigured to support each unique embedded configuration.

Ada 286 is directly integrated into the standard Intel 80286 software development environment as shown below.

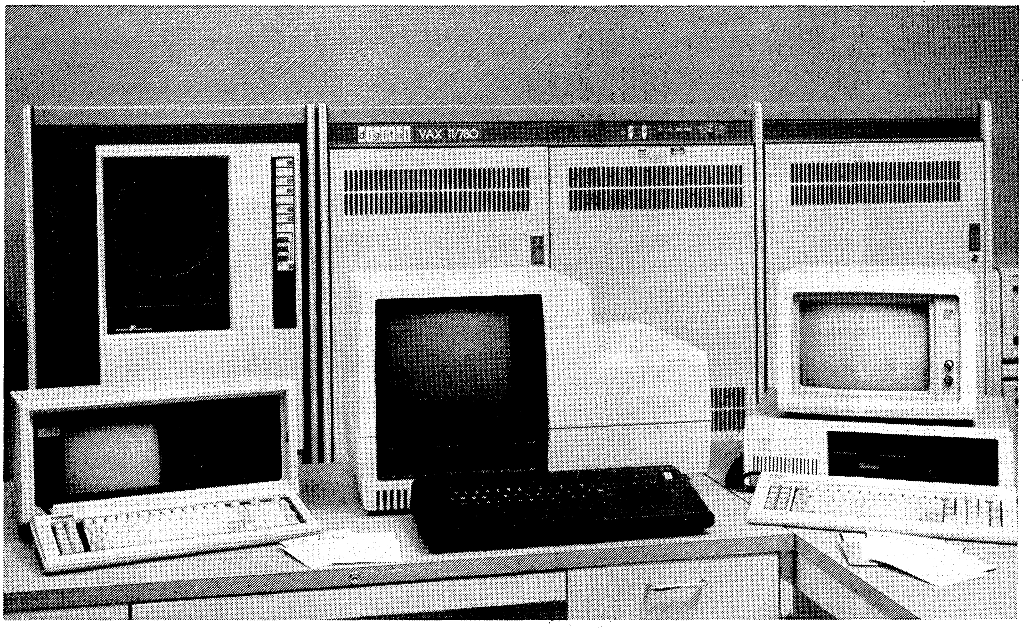
*Please note:
Ada is a trademark of the Dept. of Defense
VAX is a registered trademark of Digital Equipment Corporation

ORDER NUMBER: 231672-001


FORTRAN 286 PROGRAMS
PASCAL 286 PROGRAMS
PLM 286 PROGRAMS
ASM 286 PROGRAMS
ADA* 286 PROGRAMS
iAPX 286 BINDER
APPLICATION SOFTWARE
OPERATING SYSTEM SOFTWARE
iAPX 286 SYSTEM BUILDER
DEBUGGER ICE™, MONITOR, ETC.
PROTECTED, MULTI-TASK SYSTEM
TARGET SYSTEM

INTEL MICROPROCESSOR LANGUAGES

- Wide choice of mature languages: ASM, PL/M, C, Pascal, FORTRAN
- Optimized for Intel microprocessor architectures
- Available on industry-standard hosts: VAX/MicroVAX-VMS* and PC-DOS
- Optimized for embedded software development
- High-level symbolic debugging with Intel emulators
- Worldwide support



OCTOBER 1986
ORDER NUMBER: 280330-001

A wide variety of mature, professional languages

Intel supports each of its microprocessors and microcontrollers with a set of high-level languages specifically designed to take advantage of that component's features and performance. Whether you're designing with the 8086, the 80286, 80386, or a member of the 8051 or 8096 families, there's a wide variety of language tools available to you, from assembler to PL/M, Pascal, FORTRAN or C.

What makes Intel languages so special? They've been tuned for peak performance on Intel microprocessors, resulting in better code quality. No other language vendor can claim that. Also, they've been around for a long time, used in thousands of software development labs around the world. They're fully debugged, mature, stable. And they're supported by one of the most reputable names in the microprocessor development business: Intel.

We're committed to making our customers successful with our products, so we provide extensive training, on-site application support, telephone hot-line support, user program libraries and programming tips.

Optimized for embedded software development

Intel languages are designed for the professional software developer, particularly those doing embedded software development: programming close to the hardware, such as designing operating systems, real-time factory automation systems and other applications requiring extremely fast, compact code. Intel compilers support the special requirements of embedded software development such as ROMability, fast interrupt handling and library reentrancy, and code optimization.

Intel's PL/M is particularly well suited to microprocessor development. It was the first high-level language designed expressly for microprocessors and is still one of the most widely used tools in the microprocessor and microcontroller world. It rivals assembly language in efficiency, and is 25-50% more efficient than C.

Because all Intel languages use the same object module format, modules written in different languages (such as C or PL/M) can be linked together into a single executable program. Or, modules written on a PC can be linked with modules written on a VAX/VMS*.

Also, all Intel languages provide full symbolic information for use with Intel debuggers and emulators, such as I²ICE™, ICE™ 5100, VLSiCE 96, and PSCOPE. Intel's TYPDEF, DEBSYM and LOCSYM records contain a wealth of information about external and public variables, code blocks, local symbols and debugging symbols. Such information allows symbolic and high-level langauge debuggers to provide additional services to programmers, such as displaying local variables or producing structure dumps.

Compilers that fit your environment

Intel compilers are not only powerful, they're flexible, too. Intel compilers are available on the industry hosts of choice: the VAX (including MicroVAX), and the IBM PC and compatibles, allowing you to better allocate hardware resources. DOS tools can be used for interactive program development, and VMS tools for batch compilation and source management.

The VAX-hosted versions of Intel compilers are tailored to the VMS environment: they follow the DCL command invocation style and use "VMS Intall" installation procedures, with thorough on-line help. All Intel compilers have been optimized in VMS environments for fast I/O performance.

Using Intel networking protocols such as NDS II and OpenNET™, PCs and VAXs can be linked together for maximum productivity in the development lab.

Hardware, software support in one call

There's security in designing with Intel microprocessors and languages: you know that all your development questions will be answered by Intel's worldwide staff of trained hardware and software engineers. Our support includes maintenance, consulting and training for both VMS and DOS-hosted tools.

*VAX and VMS are trademarks of Digital Equipment Corporation.
OpenNET and Insite are trademarks of Intel Corporation.

intel®

ARTICLE
REPRINT

AR-59

June, 1978

Modular Programming in PL/M

William Brown
Microcomputer Systems Division

280324-1

Modular Programming in PL/M*

William Brown
Intel Corporation

Various methodologies have been used to control the high—and rising—cost of developing software products. Among these, one technique that has proved effective entails constructing programs from small, well-defined modules. This technique, called modular programming, can be used in any programming language; however, without language support to enforce module boundaries, errors often occur.

The PL/M language and compiler are designed to bring the advantages of modular programming to microprocessor software systems. Since the fundamental PL/M language facility for organizing a program is the module, software systems can be partitioned into manageable units. The PL/M module can hold data and procedures and, if properly used, provide encapsulation of programming abstractions. In this way it is related to several other language mechanisms that provide for grouping operations logically related to a single data structure—for example, the Simula class,[1] the Alphard form,[2] the CLU cluster,[3] and the Mesa module.[4]

Modularity

The basic motivation for modularizing a software system is to divide the system into partitions understandable to the implementer. There are many techniques for designing a partitioning. The oldest one applies a functional decomposition of the system into subroutines or procedures. However, in truly large systems, such decomposition usually results in a large number of procedures which, though easily understood, have complex interdependencies.

Encapsulation. Another technique, suggested by Parnas,[5] is based on encapsulation of information. A software system is partitioned in terms of the abstractions which make it most understandable. Thus, a text editor might be expressed as manipulations of strings or a logic simulation as a structure of logic cells. By encapsulating, or hiding, the implementation details of the abstraction, interdependencies are limited to the properties of the abstraction (for example, concatenate, find, etc., for strings, or inputs and outputs for logic cells). Thus, the system is more understandable.

Hiding information also enhances the long-term utility of the system by making programs easier to maintain and modify. First, the source text is encapsulated so that any program changes are localized. Second, if the engineering requirements of the system change, the implementation of the abstraction can be replaced without affecting any other part of the system. For example, the implementation of logic cells might initially be optimized for minimum memory-space requirements. Later, if speed becomes important, the implementation can be replaced by one optimized for speed.

PL/M modules share two aspects of encapsulation with the facilities of Alphard, CLU, and Mesa. First, the module localizes the source text which implements the abstraction. Second, the module hides implementation details. It thereby provides a certain amount of protection.

The PL/M system

This description of the PL/M language and the software development environment concentrates on those features important to modular programming. It is intended to provide enough background so that someone familiar with similar languages and systems can understand the examples. For further information, Intel's *PL/M-80 Programming Manual*[6] provides a complete description of the language, and McCracken[7] provides a tutorial introduction to

* Adapted from a paper presented at COMPSAC 77, Chicago.

PL/M and the ISIS-II diskette operating system. Intel's *ISIS-II System User's Guide*[8] describes the file management services and general facilities of this operating system.

PL/M is a block-structured procedural language. It is intended as a system implementation language for the Intel 8080 microprocessor. Syntactically, it closely resembles XPL[9] or PL/I.[10] However, the statement structure should be understandable to anyone familiar with a block-structured language.

The data types which PL/M manipulates are probably not familiar to some readers. PL/M has only two basic data types: BYTE and ADDRESS. A BYTE is an 8-bit unsigned value. An ADDRESS is a 16-bit unsigned value. In addition to these data types, PL/M allows singly dimensioned arrays and single-level data structures.

An example declaration for a BYTE variable (CH) and two ADDRESS variables (*B*1 and *B*2) is given below:

```
DECLARE CH BYTE,
        (B1, B2) ADDRESS;
```

PL/M takes a primitive approach to the problems presented by references to objects. A reference to an object is simply the memory address of the object. PL/M uses a dot to denote the operation "address of." Thus, ".CH" yields the address of "CH."

PL/M also allows for accessing variables by their references. This is provided by the BASED notation in declarations. For example, with the declaration

```
DECLARE B ADDRESS,
        CH BASED B BYTE,
        N          BYTE;
```

and the assignments

```
B = .N;
CH = 5;
```

the value of *N* is 5.

The BASED variable concept is important to the procedure mechanism. Only objects of type BYTE or ADDRESS may be passed to a procedure and all parameters are passed by value. Therefore, to pass a large object like an array or to implement a return parameter requires a BASED declaration. In this fashion, PL/M implements call by reference.

The last facility to be discussed is the LITERALLY declaration. A LITERALLY defines a parameterless macro or string substitution in the source text. Thus, with the declaration

```
DECLARE ZERO LITERALLY '0';
```

the appearance of the identifier ZERO is equivalent to writing the constant 0.

PL/M modules. A module is a labeled block which is not enclosed in any other block. Data objects and procedures can be declared in the module, and in one distinguished module (the main program module) an executable statement sequence may appear. Since a module is a block, names declared in it are normally limited to the extent of the block. Thus, all objects are *a priori* hidden inside the module. However, PL/M's PUBLIC and EXTERNAL attributes provide mechanisms to make names in one module explicitly visible in another. (This formulation parallels the Mesa facilities.)

A procedure or data object in a module may be given the PUBLIC attribute. This makes the name of the object visible outside the module. Only objects declared at the first nesting level may be declared PUBLIC. This restriction, and the fact that modules are statically allocated, assures that PUBLIC procedures have a consistent environment for efficient execution.

A module may access PUBLIC information in another module by including a matching EXTERNAL declaration. For a procedure, the EXTERNAL declaration appears as a procedure with only parameter declarations in the body. The attribute EXTERNAL appears as the last item in the procedure head. For data, PUBLIC or EXTERNAL appears as an attribute in the declarations. For example, the declaration

```
DECLARE NAMEREC STRUCTURE(
                LAST    (25) BYTE,
                FIRST   (25) BYTE,
                MI           BYTE) PUBLIC;
```

declares a structure variable, NAMEREC, which has three fields. The fields LAST and FIRST are arrays of 25 BYTES. The field MI is a single BYTE. The matching EXTERNAL declaration is

```
DECLARE NAMEREC STRUCTURE(
                LAST    (25) BYTE,
                FIRST   (25) BYTE,
                MI           BYTE) EXTERNAL;
```

The names of structure fields and procedure parameters in EXTERNAL declarations need not match those in the PUBLIC declaration. Only the types and order must match.

The compiler and linkage system. The current PL/M compiler has two features which are important to implementing modular abstractions. First, the module is the natural unit of compilation. Thus, an implementation of an abstraction can be compiled once and then used for many applications. Second, the compiler supports a textual inclusion facility. This facility is provided by a compiler control having the following general form

```
$INCLUDE (filename)
```

The compiler will read the file given by the filename. The text read will be inserted into the source program, replacing the INCLUDE control. The

EXTERNAL and LITERALLY declarations for a module may be included this way. Thus, an abstraction may be referenced by a single name. Textual inclusion is the mechanism used by Mesa for static binding of implementations of an abstraction to users of the abstraction.

The linkage system is responsible for binding modules together. It matches all EXTERNAL declarations to the appropriate PUBLIC declarations. Unfortunately, this matching is done by name only. No type checking is performed.

Example abstraction—strings. The abstraction to be implemented is that of variable-length character strings. The abstraction has the following operations: LENGTH, COPY, CONCAT, FRONT, REST, FIND, BLANKS, PUT, and GET. It is possible to define each of these operations in precise mathematical terms. However, for the purpose of this example, only informal descriptions with a minimum of formal notation are given. Where a functional notation is necessary, *S* will represent a string and *N* will represent a non-negative integer.

LENGTH returns the number of characters in the argument string. The empty string has a length of zero.

COPY returns a duplicate of the argument string.

CONCAT returns a string which is a concatenation of its arguments. The order of concatenation is the first argument string followed by the second. The two argument strings are not affected.

FRONT returns a string which is a copy of the first *N* characters of the argument string. The value of *N* must be in the inclusive range from 0 to the length of the string. If *N* is zero an empty string is returned.

REST returns a string such that CONCAT (FRONT(S,N), REST(S,N)) is a copy of the string *S*.

FIND locates a character in the argument string and returns the length of the substring ended by that character. If the character is not in the string, zero is returned.

BLANKS returns a string of blanks of a specified length. BLANKS (0) returns an empty string.

PUT outputs a string as a line on a specified file.

GET inputs a line from a specified file and converts it to a string.

```
Declare Ref$String Literally 'Address',
        Character Literally 'Byte';

Length:
  Procedure (Ref) Address External;
    Declare Ref Ref$String;
  End Length;

Blanks:
  Procedure (N) Ref$String External;
    Declare N Address;
  End Blanks;

Copy:
  Procedure (Ref) Ref$String External;
    Declare Ref Ref$String;
  End Copy;

Concat:
  Procedure (Ref1, Ref2) Ref$String External;
    Declare (Ref1, Ref2) Ref$String;
  End Concat;

Front:
  Procedure (Ref, Ind) Ref$String External;
    Declare Ref Ref$String,
            Ind Address;
  End Front;

Rest:
  Procedure (Ref, Ind) Ref$String External;
    Declare Ref Ref$String,
            Ind Address;
  End Rest;

Find:
  Procedure (Ref, Ch) Address External;
    Declare Ref Ref$String,
            Ch Character;
  End Find;

Put:
  Procedure (Ref, Fl) External;
    Declare Ref Ref$String,
            Fl Address;
  End Put;

Get:
  Procedure (Fl) Ref$String External;
    Declare Fl Address;
  End Get;

Delete:
  Procedure (Ref) External;
    Declare Ref Ref$String;
  End Delete;
```

Figure 1. The user's view of strings defined by external declarations.

The implementation

Before implementing the string abstraction, concrete PL/M interfaces for the abstract operations must be specified. Figure 1 contains the EXTERNAL and LITERALLY declarations which define strings to the user. These declarations correspond to a definition module in Mesa or the specification part of an Alphard form. To produce these declarations two implementation details had to be fixed.

First, since PL/M allows only scalar parameters, the concept of "references to a string" has been introduced. The LITERALLY declaration defines REF$STRING as ADDRESS. This does not imply, however, that a reference to a string is necessarily the memory address of the representation. The actual representation of the object is hidden by the module structure. This LITERALLY provides for visually distinguishing declarations of string references from other variables of type ADDRESS. However, the language does not enforce any distinction.

Second, an additional operation, DELETE, has been specified. The abstraction was not concerned with the problem of dynamic storage management. It is possible to implement strings with implicit storage management. However, that would complicate the representation. Therefore, the user is responsible for deleting unused strings.

Representation. The user's view of strings is defined by the declarations in Figure 1. These declarations do not imply anything about the represeentations of strings or string references; the module structure is used to hide these details. Several alternatives are possible. A string might be represented as a linked list of characters or as a dynamically allocated BYTE array. String references might be the address of the string representation or an index into a hidden array maintained by the module.

The representation chosen implements a string reference as the address of a dynamically allocated BYTE array. However, to illustrate encapsulation and the effect of engineering decisions on an implementation, two forms of this representation are supported. For strings of less than 255 characters, the first entry in the dynamic array is the length of the string. Thus, short strings are handled efficiently in minimum space. For strings of 255 or more characters, the first entry in the dynamic array is 255 and the end of the string is indicated by another 255. Thus, long strings pay a slight penalty in both space and time. If a more efficient representation for long strings is required, the representations can be changed without impacting the user of the abstraction.

Completed module. The source text for the completed module to implement strings is in the appendix. This module corresponds to a program module in Mesa or the representation and implementation parts of an Alphard form. The implementation is not completely representative of good software development in that the source text is not adequately documented and it has been validated only to the extent necessary to run the example.

Notice that the STRINGS module accesses two other abstractions by INCLUDE. The first of these provides EXTERNAL declarations for the ISIS-II input/output facilities, described in the user's guide.[8] The second abstraction, referenced by the file name MEMMAN.DEF, provides for dynamic storage management. This module contains two operations, ALLOC and DEALLOC, which allocate and deallocate contiguous blocks of memory.

The module contains several useful LITERALLY declarations. In addition to REF$STRING and CHARACTER declarations, the type STRING is declared literally. Since this type is always applied to BASED items, the array length specifier of 1 is only a formality.

The procedure NEW is hidden inside the STRINGS module. It takes as a parameter the length of a string to be created and allocates space for the appropriate representation type. It also initializes the length or boundary markers.

The PUBLIC procedure LENGTH defines the length operation. It is typical of the procedures implementing the operations. The first line names the procedure and formal parameter, and the word ADDRESS indicates this is a function returning an ADDRESS value. The word PUBLIC indicates the procedure is to be accessible outside the module. Next comes the declaration of the parameter and two local variables. The first is a STRING based on the reference parameter. The second is a counter for a loop. The body of the LENGTH procedure follows.

The remaining procedures follow the same pattern. However, two points should be mentioned. First, several procedures call MOVE, a built-in PL/M procedure for moving bytes from one memory area to another. Second, the DELETE procedure does not free all the storage for unused strings. The length of the string is set to zero and the remaining storage is freed. This action helps avoid problems arising from inadvertently referencing a deleted string. It is, of course, hidden from the user of the abstraction.

Example program. Figure 2 shows a program using the string abstraction. The input to this program is a text file, TEST.SRC, containing tab characters. Tabs are represented in the text by the character '/'. The program processes the file and outputs the text file TEST.OUT. The output has the tab characters replaced by enough blanks to implement tab stops at columns 8, 16, 24, 32, etc.

The INCLUDES of the files IO.DEF and STRING.DEF at the beginning of the program supply the EXTERNAL declarations for the abstractions. The text of STRING.DEF is exactly that given in Figure 1. The text of IO.DEF is described in the discussion of the module STRINGS.

Next is the procedure declaration for CONCATD. This declaration provides a local extension to the string abstraction. It implements a concatenation operation which deletes the argument strings. Note that this extension is defined in terms of the operations of the string abstraction, and not in terms of the actual representation. Thus, the encapsulation of the implementation is preserved.

Following the procedure declaration are the declarations for the variables used by the program. The variables LINE and OUTLINE are references to the input string and output string, respectively. The rest of the variables are various temporaries and counters.

The body of the algorithm is an iteration which terminates when a null string is encountered. Each LINE is processed in turn until all tabs have been found. When a tab is found (by FIND), all the characters in the line in front of the tab are concatenated to the output string (referenced by OUTLINE). Next, the length of this new string is determined and the proper number of blanks to be inserted is calculated (as LB). This number of blanks is concatenated to the output string. Finally, the original string LINE is replaced by the REST of the string and a new tab is located.

When no more tabs are found, the remaining part of the input string is concatenated to the output string. This string is output. A new LINE is input and the outer iteration is repeated.

```
Tabs:Do:

$Include (String.Def)
$Include (Io.Def)

  Concatd:
    Procedure (Ref1,Ref2) Ref$String:
      Declare (Ref1,Ref2,Retref) Ref$String:
        Retref = Concat (Ref1,Ref2):
        Call Delete (Ref1):
        Call Delete (Ref2):
        Return Retref:
    End Concatd:

  Declare (Line,Outline,Tmp) Ref$String,
        (I,L,Lb) Address,
        (Infile,Outfile,Status) Address:

  Declare Tab Literally '''/''':

  Call Open
    (.Infile,.('TEST.SRC '),1,256,.Status):
  Call Open
    (.Outfile,.('TEST.OUT '),2,0,.Status):

  Line = Get(Infile):
  Do While Length(Line) <> 0:
    Outline = Blanks(0):
    I = Find(Line,Tab):
    Do While I <> 0:
      Outline =
        Concatd(Outline,Front(Line,I-1)):
      L = Length(Outline):
      Lb = (((L/8)+1)*8)-(L+1):
      Outline =
        Concatd(Outline,Blanks(Lb)):
      Tmp = Line:
      Line = Rest(Line,I):
      Call Delete(Tmp):
      I = Find(Line,Tab):
    End:
    Outline = Concatd(Outline,Line):
    Call Put(Outline,Outfile):
    Call Delete(Outline):
    Line = Get(Infile):
  End:

  Call Exit:
End Tabs:
```

Figure 2. Example program using the string abstraction.

Figure 3 shows an input file and the corresponding output file. The output was obtained by supplying a reasonable implementation of the memory management module and executing the TABS program.

```
count/amount/total
25/$.25/$6.25
5/$.42/$2.10
7/$3.20/$22.40

count    amount    total
25       $.25      $6.25
5        $.42      $2.10
7        $3.20     $22.40
```

Figure 3. Input file with the corresponding output file.

Conclusion

As the example program shows, the PL/M module is a simple, efficient encapsulation mechanism that can emulate many of the abstraction facilities of Alphard, Mesa, and CLU. Thus, a number of benefits inherent in such languages, including better readibility and maintainability, are available to the PL/M programmer. Discipline is required, however, since existing implementations of PL/M—unlike those of the other languages—do not check for consistent use of abstractions.

The language facilities and methodology exemplified by the STRINGS module can be successfully applied to real software products. They have been used, for example, in constructing the foundation of Intel's RMX-80 real-time operating system which coordinates programs performing real-time control functions.[11] ■

Acknowledgments

I wish to thank Kevin Kahn and John Doerr for their many comments and suggestions during the writing of this article.

Appendix. Source text for the completed module which implements the strings example.

```
Strings:Do:

$Include (Io.Def)
$Include (Memman.Def)

  Declare Ref$String  Literally 'Address',
          String      Literally '(1) Byte',
          Character   Literally 'Byte',
          Cr          Literally '13',
          Lf          Literally'10':

  New:
    Procedure (Ln) Ref$String:
      Declare Ln Address,
              Retref Ref$String,
              Str Based Retref String:

        If Ln >= 255 Then Do:
          Retref = Alloc(Ln+2):
          Str (0),Str(Ln+1) = 255:
        End: Else Do:
          Retref = Alloc(Ln+1):
          Str (0) = Ln:
        End:
        Return Retref:
    End New:

  Length:
    Procedure (Ref) Address Public:
      Declare Ref Ref$String,
              Str Based Ref String,
              I Address:

        If Str (0) < 255 Then Return Str (0):
        I = 1:
        Do While Str (I) <> 255:
          I = I+1:
      End:
      Return (I-1):
  End Length:
```

```
Blanks:
  Procedure (N) Ref$String Public;
    Declare (N,I) Address,
            Retref Ref$String,
            Str Based Retref String;

      Retref = New(N);
      If N <> 0 Then
        Do I = 1 To N;
          Str(I) = ' ';
        End;
      Return Retref;
End Blanks;

Copy:
  Procedure (Ref) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            Ln Address;

      Ln = Length(Ref);
      Retref = New(Ln);
      If LN <> 0 Then
        Call Move(Ln,Ref+1,Retref+1);
      Return Retref;
  End Copy;

Concat:
  Procedure (Ref1,Ref2) Ref$String Public;
    Declare (Ref1,Ref2,Retref) Ref$String,
            (Ln1,Ln2) Address;

      Ln1 = Length(Ref1);
      Ln2 = Length(Ref2);
      Retref = New(Ln1+Ln2);
      If Ln1 <> 0 Then
        Call Move(Ln1,Ref1+1,Retref+1);
      If Ln2 <> 0 Then
        Call Move(Ln2,Ref2+1,Retref+Ln1+1);
        Return Retref;
  End Concat;

Front:
  Procedure (Ref,Ind) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            Ind Address;

      Retref = New(Ind);
      If Ind <> 0 Then
          Call Move(Ind,Ref+1,Retref+1);
      Return Retref;
  End Front;

Rest:
Procedure (Ref,Ind) Ref$String Public;
    Declare (Ref,Retref) Ref$String,
            (Ln,Restln,Ind) Address;

      Ln = Length(Ref);
      Restln = Ln-Ind;
      Retref = New(Restln);
      If Restln <> 0 Then
        Call Move(Restln,Ref + Ind + 1,Retref +1);
      Return Retref;
  End Rest;

Find:
  Procedure (Ref,Ch) Address Public;
    Declare Ref Ref$String,
            Str Based Ref String,
            Ch Character,
            (Ln,I) Address;

      Ln = Length(Ref);
      If Ln = 0 Then Return 0;
      I = 1;
      Do While I <= Ln and Str (I) <> Ch;
        I = I+1;
      End;
      If Str (I) = Ch Then Return I;
      Return 0;
    End Find;

  Put:
    Procedure (Ref,Fl) Public;
      Declare Ref Ref$String,
              (Fl,Ln,Status) Address;

        Ln = Length(Ref);
        If Ln <> 0 Then
          Call Write(Fl,Ref + 1,Ln,.Status);
        Call Write(Fl,.(Cr,Lf),2,.Status);
    End Put;

Get:
  Procedure (Fl) Ref$String Public;
    Declare Retref Ref$String,
            (Fl,Actual,Status) Address,
            Buffer(128) Byte;

      Call Read
        (Fl,.Buffer,128,.Actual,.Status);
      If Actual = 0 then Return New(0);
      Retref = New(Actual-2);
      Call Move(Actual-2,.Buffer,Retref + 1);
      Return Retref;
  End Get;

  Delete:
    Procedure (Ref) Public;
      Declare Ref Ref$String,
              Str Based Ref$String;

        Call Dealloc(Ref + 1,Length(Ref));
        Str (0) = 0;
    End Delete;

  End Strings;
```

March 1978.

References

1. O. J. Dahl, B. Myhrhaug, and K. Nygaard, *The SIMULA 67 Common Base Language*, Publication S-22, Norwegian Computing Center, Oslo, 1970.

2. A. Wulf, "ALPHARD: Toward a Language to Support Structured Programming," Carnegie-Mellon University Tech Report AD-785417, April 1974.

3. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4, April 1974, pp. 50-59.

4. C. M. Geschke, J. H. Morris, Jr., and E. H. Satterthwaite, "Early Experience with Mesa," *CACM*, Vol. 20, No. 8, August 1977, pp. 540-552.

5. D. Parnas, "A Technique for Software Module Specification," *CACM*, Vol. 15, No. 5, May 1972, pp. 330-336.

6. Intel Corp., *PL/M-80 Programming Manual*, Document No. 98-268B, 1977.

7. D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*, Addison-Wesley Publishing Co., Reading, Mass., 1978.

8. Intel Corporation, *ISIS-II System User's Guide*, Document No. 98-306A, 1976.

9. W. M. McKeeman, J. J. Horning, and D. B. Wortmann, *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

10. ANS Committee X3, *Draft Proposed Standard Programming Language PL/I*, February 1975.

11. Kevin Kahn, "A Small-Scale Operating System Foundation for Microprocessor Applications," *Proc. IEEE*, Vol. 66, No. 2, February 1978, pp. 75-89.

William L. Brown is a senior software engineer in Intel's Microcomputer Systems Division in Aloha, Oregon. His past work includes the revision and enhancement of the PL/M language and the development support software for Intel's bit slice processor. He received his MEE from Rice University in 1974. He is currently an active member of the ACM and the IEEE Computer Society.

280324-7

ARTICLE
REPRINT

AR-136

June 1980

PL/M-86 Combines Hardware Access With High-Level Language Features

Mary Jane Elmore
Electronic Design, April 26, 1980

ORDER NUMBER:451145-001

451145–1

PL/M-86 combines hardware access with high-level language features

PL/M-86, a systems-implementation language, is the first high-level language (HLL) designed specifically for the special requirements of microcomputers. The user gets not only high-level access to the µP hardware, and thus control over the processor and its peripheral components, but also such HLL advantages as the ability to write code in English-like statements, more efficient software design and easier debugging and maintenance. Major features include:

- High-level constructs for machine control, especially interrupt handling, direct-port I/O and access to absolute memory locations
- Pointers and based variables
- String manipulation
- LOCKSET, a procedure for multiprocessing environments.

Designed to be executed by Intel's 16-bit 8086 (ELECTRONIC DESIGN, March 1, 1980, p. 97), PL/M-86 is upward-compatible with PL/M-80. Except for interrupts, hardware flags and time-critical code sequences, PL/M-80 programs may be recompiled under PL/M-86 with little or no conversion.

Block-structured language

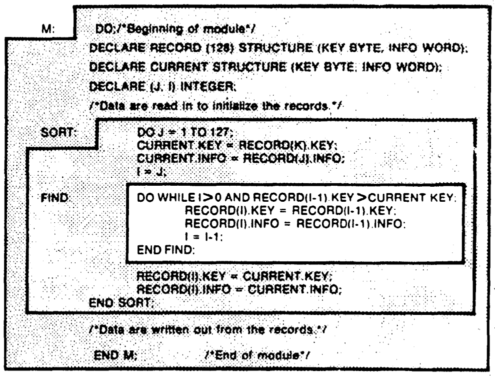
Both versions are block-structured, encouraging a structured approach to programing with well-structured branching and control statements. They provide a DO-END construct for simple block structures, as well as DO WHILE, DO CASE, an iterative DO, binary decision mechanisms IF-THEN-ELSE and nested IF-THEN-ELSE.

PL/M-86 procedures isolate well-defined tasks where local variables, valid only within their procedure, can be used to avoid unwanted interactions between procedures (Fig. 1). By making it easy to divide the programming tasks into subtasks, PL/M-86 encourages top-down design and permits several software designers to work in parallel. Since programs under development tend to keep changing, modularity also simplifies program maintenance. With PL/M-86, programs can be designed in such a way that one program function can be modified without unexpected repercussions elsewhere in the program.

In addition, as an SIL, PL/M-86 includes special features for writing systems software: I/O handlers, device drivers, system monitors—in short, any executive program that directly controls hardware, even if imbedded in application software (for instance, in machine or instrument control).

An SIL like PL/M-86 allows the system designer to control hardware with HLL constructs rather than error-prone assembly language. Specifically, the system designer can write interrupt-handling routines and routines to input or output data directly to CPU ports. PL/M-86 also allows the programmer to access memory locations directly and provides a flexible means of manipulating data and procedure pointers. Built-in procedures give access to the hardware stack pointer and CPU flags.

Unlike application-oriented languages, PL/M-86



```
M:      DO;/*Beginning of module*/
        DECLARE RECORD (128) STRUCTURE (KEY BYTE, INFO WORD);
        DECLARE CURRENT STRUCTURE (KEY BYTE, INFO WORD);
        DECLARE (J, I) INTEGER;
        /*Data are read in to initialize the records.*/
SORT:           DO J = 1 TO 127;
                CURRENT.KEY = RECORD(K).KEY;
                CURRENT.INFO = RECORD(J).INFO;
                I = J;
FIND:           DO WHILE I>0 AND RECORD(I-1).KEY >CURRENT KEY;
                        RECORD(I).KEY = RECORD(I-1).KEY;
                        RECORD(I).INFO = RECORD(I-1).INFO;
                        I = I-1;
                END FIND;
                RECORD(I).KEY = CURRENT.KEY;
                RECORD(I).INFO = CURRENT.INFO;
        END SORT;
        /*Data are written out from the records.*/
         END M;          /*End of module*/
```

1. Three nested blocks illustrate block hierarchy: Block M includes the whole screened area; block Sort includes all the code with medium and light screen; block Find is outlined by the white area only.

451145-2

lets the programmer interface directly with the system hardware, without having to bring additional modules in at execution time to interface with the hardware. While Pascal or Fortran requires an operating system or run-time support to perform system-level functions, PL/M-86's "bare-machine" programming saves memory, as the code overhead for an operating or run-time system is eliminated. This SIL thus offers the best of two worlds—the memory efficiency of system-level code and the programming efficiency of an HLL.

```
              •

              •

              •
HITEMP:  PROCEDURE  INTERRUPT(5):
           DECLARE  INTERRUPT$ID      BYTE.
                    INDEX. OUTDEX     BYTE.
                    CURRENT$STATUS  WORD:
                       •

                       •
           INTERRUPT$ID = INPUT(INDEX):

           IF INTERRUPT$ID = 00000001B   THEN
           DO:
              OUTPUT(OUTDEX) = 11000000B /*ALARM AND SHUTDOWN*/
              OUTDEX = OUTDEX + 1
              GO$FLAG + FALSE
           END:
           IF INTERRUPT$ID + 00001000B   THEN
           DO:
              OUTPUT(OUTDEX) = 10000000B   /*WARNING LIGHT*/
              OUTDEX = OUTDEX +1
           END:
           ELSE DO:
                      •

                      •

           END        •

END HITEMP:
```

2. Although a high-level language, PL/M-86 provides direct access to hardware. In this example, a peripheral signals INTERRUPT(5) whenever a certain temperature exceeds its limit. The shown interrupt procedure activates warning signals and stops the process.

```
SORT:   DO J = 1 TO COUNT-1;
             CALL MOVB (@RECORD(J*RECSIZE),@CURRENT.RECSIZE):
             I=J.

FIND:        DO WHILE 1      0
                                   AND RECORD (1-1)*RECSIZE + KEY
                                     CURRENT(KEY):
                  CALLMOVB(@RECORD(1-1)*RECSIZE).
                                  @RECORD(1*RECSIZE).
                                  RECSIZE:
                  1=1-1:
             END FIND:

             CALL MOVB(@CURRENT.@RECORD(1*RECSIZE).RECSIZE):
        END SORT:
```

3. In this fragment from a SORT routine, the predefined procedure MOVB is called several times. Being a built-in procedure, it does not have to be declared. In the first call (highlighted), the parameter @ RECORD(J*RECSIZE) specifies the starting address of the byte sequence to be copied; @ CURRENT is the location to which the first byte will be copied; RECSIZE is the number of bytes in the stream of data to be transferred.

Interrupts make it possible to break into the execution sequence of a running program to carry out other tasks and then resume execution of the interrupted program. Sometimes, the external event is repetitive —for instance, a clock pulse that only needs to be counted before other processing resumes. At other times, the external event can be a signal indicating that data are ready to be input or that some process has exceeded allowable limits.

Since μC applications involve processing of interrupts to some degree, an SIL must include provisions for interrupt-handling routines. In an 8086-based system, an interrupt may be generated by some peripheral device that sends an interrupt signal and number to the 8086 CPU (Fig. 2).

The CPU processes an interrupt by:

- Completing the machine instruction currently under execution
- Disabling the interrupt mechanism
- Activating an interrupt procedure corresponding to the number sent by the peripheral device.

After executing a RETURN or END statement, the interrupt procedure automatically reenables the interrupt mechanism and returns control to the point where the interrupt occurred.

For I/O operations, PL/M-86 provides built-in procedures that let the programmer access the CPU's I/O ports directly. This includes support for byte or word I/O and constant or variable port numbers. To input a byte from an 8086 I/O port, use

INPUT (expression)

The value of "expression" specifies one of the input ports of the 8086 CPU. The value returned by INPUT is the byte value found in the specified input port (see Fig. 2).

To access specific memory locations, PL/M-86 provides the AT attribute:

AT (location)

where "location" may be either a whole-number constant in the range of 0 through 1,048,575 or a location reference. The latter uses the "@ operator" to indicate where a specific variable will reside at execution time. For example, @ RESULT represents the run-time location of the variable RESULT. The statement

DECLARE (CHAR$A, CHAR$B) BYTE AT (4096);

causes the BYTE variable CHAR$A to be stored at location 4096. The variable CHAR$B follows in the next two bytes.

On the other hand, the construct

451145-3

```
DECLARE DATUM WORD
DECLARE ITEM BYTE AT (@DATUM)
```

causes ITEM to be declared a BYTE variable, located at the location of DATUM. PL/M-86's ability to access absolute memory locations is especially important for memory-mapped I/O or other hard-wired memory locations.

What are based variables?

Sometimes a direct reference to a variable is either impossible or inconvenient—for example, when the location of a data element remains unknown until it is computed at run time. It may then be necessary to manipulate the locations of data elements rather than the data elements themselves. PL/M-86 provides this indirect form of reference with "based variables." The base of a based variable is another variable pointing to the based variable. Both must be declared separately, with the base coming first. For instance, in

```
DECLARE ITEM$PTR POINTER;
DECLARE ITEM BASED ITEM$PTR BYTE;
```

ITEM$PTR is base and ITEM is the based variable. The construct

```
ITEM$PTR=34AH;
ITEM     =  77H;
```

loads the value 77 (hex) into the memory location 34A (hex).

One variable name can refer to many different data

```
        .
        .
DECLARE  B BYTE, C CBYTE,
  TEST  BYTE,
  A WORD;
IF  TEST  THEN
DO;
  OUTWARD (0F6H)-0FFFFH;
  A=B
END;
ELSE A=C
```

```
            MOV     AL,TEST
            RCR     AL,1
            JB      @1
            JMP     @2

    @1:     MOV     AX,0FFFFH
            OUTW    0F6H

            MOV     AL,B
            MOV     AH,0H
            MOV     A,AX
            JMP     @3

    @2:     MOV     AL,C
            MOV     AH,0H
            MOV     A,AX

    @3:
```

(a)

```
            MOV     AL,TEST
            RCR     AL,1
            JB      @1
            JMP     @2

    @1:     MOV     AX,0FFFFH
            OUTW    0F6H
            MOV     AL,B
            JMP     @4
            MOV     AH,0H
            MOV     A,AX
            JMP     @3

    @2:     MOV     AL,C
    @4:     MOV     AH,0H
            MOV     A,AX

    @3:
```

(b)

```
            MOV     AL,TEST
            RCR     AL,1
            JB      @1
            JMP     @2

    @1:     MOV     AX,0FFFFH
            OUTW    0F6H
            MOV     AL,B
            JMP     @4

    @2:     MOV     AL,C
    @4:     MOV     AH,0H
            MOV     A,AX

    @3:
```

(c)

```
            MOV     AL,TEST
            RCR     AL,1
            JNB     @2

            MOV     AX,0FFFFH
            OUTW    0F6H
            MOV     AL,B
            JMP     @4

    @2:     MOV     AL,C
    @4:     MOV     AH,0H
            MOV     A,AX

    @3:
```

(d)

4. An ASM86 program—before optimization (a), after cross-jumping (b), after elimination of unreachable code (c) and after reversing a branch condition (d).

451145–4

items depending on the value of the base. For instance, the loop

```
TOTAL = 0;
DO ITEM$PTR = 2100H to 2199H;
TOTAL = TOTAL + ITEM
END;
```

places in TOTAL the sum of the 256 bytes found in memory locations 2100H through 2199H.

Based variables are even more powerful when the "@ operator" is used to supply values for bases. For example, suppose there are three different real variables, A$ERROR, B$ERROR, and C$ERROR, which should be accessible at different times via the single identifier ERROR. This can be done as follows:

```
DECLARE (A$ERROR, B$ERROR, C$ERROR) REAL;
DECLARE ERROR$PTR POINTER;
DECLARE ERROR BASED ERROR$PTR REAL;
ERROR$PTR = @A$ERROR;
```

At this point, the value of ERROR$PTR is the location of address A$ERROR. A reference to ERROR is, in effect, a reference to A$ERROR. Later in the program, the statement ERROR$PTR = @C$ERROR; turns a reference to ERROR into a reference to C$ERROR. This technique is useful not only for manipulating complicated data structures but also for passing locations to procedures as parameters.

With strings attached

One of the key features built into the 8086 is the ability to handle large-scale string-manipulation assignments far more easily than the 8080 and the 8085. PL/M-86 exploits this feature, with very powerful string-handling procedures to scan, translate or move blocks of bytes or words in ascending or descending order. The system designer thus has access to the 8086's string capabilities without having to worry about absolute memory locations and register contents, as an assembly-language programmer would (Fig. 3).

Another feature designed into the 8086 architecture is multiprocessing capability, accessible via the LOCKSET procedure. Through it, the system designer gains control over shared resources by locking other processors out while, for instance, a memory block is being updated. In a system where an 8086 processor offloads its I/O control tasks to an 8089 I/O processor, some memory locations may be used by both processors.

While the 8086 is accessing and updating that memory location, the 8086 should not be outputting data from that location or writing new data into that location. So a flag is set or reset depending on whether or not the processor seeking access to the critical resource can obtain that access.

An optimizer saves memory

Memory may be cheap, but in a large production run every byte still counts. So, an optimizing compiler will soon pay for itself. PL/M-86 uses a number of optimization techniques:

Folding of constant expressions

Calculating the value of constants in expressions at compile time rather than generating code to calculate it at run time saves both time and memory. In the expression

A = 6 + 3 + A;

the compiler will add 6 and 3 first and produce code to add 9 to A.

Strength reduction

This term applies to the replacement of certain instructions with faster, shorter ones. For example, performing a left-shift of one bit replaces a multiplication by two; n left-shifts correspond to a multiplication with 2^n.

Elimination of common expressions

If an expression appears more than once in the same block, its value is saved rather than recomputed each time. For example, in

A = B + C*D/3
C = E + C*D/3

the value of C*D/3 need not be computed a second time.

Short-jump optimization

When there's a choice of different jump-instruction types, the compiler selects the smallest one possible.

Branch optimization

Branch chaining reduces a branch to another branch to a single branch instruction:

```
BEFORE                      AFTER
   JMP LAB1                    JMP LAB2
     .                           .
     .                           .
     .                           .
LAB1: JMP LAB2              LAB1: JMP LAB2
LAB2:                       LAB2:
```

451145-5

Having defined a BYTE variable (called LOCK, for example), the LOCKSET instruction sets that variable to a value that denies memory access.

If LOCK=1 means "access not available" and LOCK=0 means "access allowed," and if all processors in the system have been programmed to recognize that convention, the following code segment gives access to a *critical* memory location while preventing other processors from doing so until the operation is finished:

```
    /*BEGIN CRITICAL REGION*/
    DO WHILE LOCKSET (@LOCK, 1);
    END;
                    .
                    .
                    .
    LOCK=0;
    /*END CRITICAL REGION*/
```

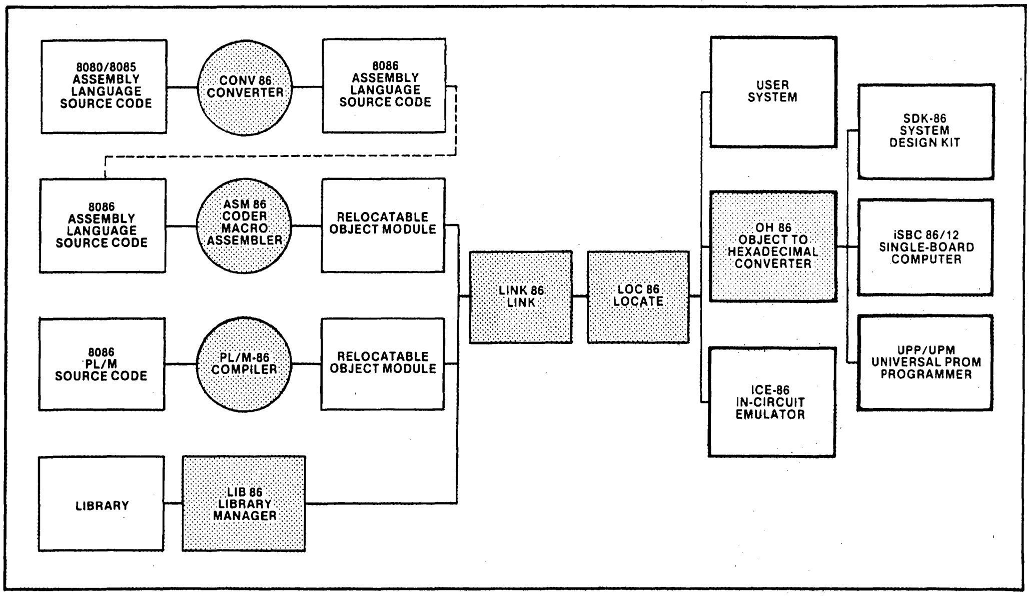
In this segment, the processor loops until memory location LOCK is reset by another processor—i.e., LOCKSET returns ZERO until that processor sets LOCK to prevent other processors from accessing the memory area. The processor carries out its program, then unlocks the memory area (LOCK=0). The first executable line of the program segment (DO WHILE...) references the variable LOCK and assigns the value 1 to that location.

If the value returned is 0, LOCK had not already been set and the current processor has now set it. But if the value returned is 1, the LOCK had already been set and the processor must wait until the busy processor releases the memory lock. Since the locking mechanism uses a simple BYTE variable, there is no practical limit to the number of locks available.

A language isn't enough

PL/M-86 is implemented as a compiler, not as an interpreter, because in the normal μC design process a debugged program is loaded into PROMs for the prototype system. A compiler produces object modules in a form that can be directly executed by the CPU.

The PL/M-86 compiler boasts many compile-time options to help with coding and debugging. Most important is *conditional compilation*, which permits the compiler to skip over selected portions of the source code if certain conditions are met. This feature enables the designer to produce different object modules for different applications of the program. An INCLUDE command, on the other hand, allows the user



8080/8085 ASSEMBLY LANGUAGE SOURCE CODE
CONV 86 CONVERTER
8086 ASSEMBLY LANGUAGE SOURCE CODE
8086 ASSEMBLY LANGUAGE SOURCE CODE
ASM 86 CODER MACRO ASSEMBLER
RELOCATABLE OBJECT MODULE
8086 PL/M SOURCE CODE
PL/M-86 COMPILER
RELOCATABLE OBJECT MODULE
LIBRARY
LIB 86 LIBRARY MANAGER
LINK 86 LINK
LOC 86 LOCATE
USER SYSTEM
OH 86 OBJECT TO HEXADECIMAL CONVERTER
ICE-86 IN-CIRCUIT EMULATOR
SDK-86 SYSTEM DESIGN KIT
iSBC 86/12 SINGLE-BOARD COMPUTER
UPP/UPM UNIVERSAL PROM PROGRAMMER

5. The PL/M-86 package (screen) contains, in addition to the compiler, an 8086 assembler and many important utilities. The final machine code can be loaded into a number of optional hardware items.

451145-6

to include routines from a different source file as well.

Another compiler option, CODE/NOCODE, provides listings of the generated object code in assembly-language format, interleaved with the PL/M statements for easier debugging. The PL/M-86 compiler also provides a flexible cross-reference of program symbols between PL/M-86 modules.

The PL/M-86 compiler also includes sophisticated code-optimization techniques to produce efficient object modules. A compile-time OPTIMIZE control provides three levels of optimization: Level 0 skips optimization for a quick compilation. Level-1 optimization is the PL/M-86 default and provides constant-folding, strength reduction and elimination of common expressions. Level 2 adds jump optimization, branch chaining, cross-jumping and deletion of unreachable code (see "An Optimizer Saves Memory").

An example incorporating several optimization techniques is shown in Fig. 4. The program determines whether the byte variable TEST is true (i.e., the least significant bit is 1). If it is, the hex value 0FFFF will be output to port 0F6H and the value of the BYTE variable B will be assigned to the WORD variable A. If the variable TEST is not true, variable A will be assigned the value C.

The assembly code produced by the short PL/M-86 module contains 57 bytes (Fig. 4a). Cross-jumping inserts a JUMP (line 8, Fig. 4b) to combine the identical code at the end of two converging paths (lines 8 and 9 and 12 and 13 in Fig. 4a) and diverts the program flow to the second occurrence of the two lines. The first occurrence is now unreachable and can be deleted (Fig. 4c). Another line of code is saved by reversing a branch condition, which produces line 3 of Fig. 4d.

The PL/M-86 compiler, which runs on Intel's Intellec μC-development system, is not a "stand-alone" design tool but part of an integrated set of design-aid tools for the 8086 or 8088. These tools include an assembler for ASM86, a high-level assembly language that produces object modules compatible with those from PL/M-86 (both can be combined using the 8086/8088 relocation and linkage tools).

ASM86 complements PL/M-86 since it lets the programmer choose the language most appropriate for a task and then combine the modules. Commonly used PL/M-86 and ASM86 object modules can be stored and managed using LIB86, the 8086 object-module librarian. PL/M-86 or ASM86 object modules may be loaded by the ICE-86 in-circuit emulator, and the software may then be debugged and integrated with the hardware. After hex conversion, Intellec's PROM programmer allows the debugged object modules to be stored in EPROMs (Fig. 5).■■

451145-7

SYSTEM DESIGN/SOFTWARE

COMPILER OPTIMIZATION TECHNIQUES

Techniques used within the PL/M-86 compiler make the programmer's job easier while supplying highly efficient code

by Armond Inselberg and Stan Mazor

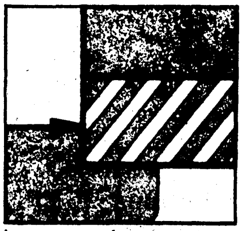
Increasing demands for software development have combined with continuing shortages of programming personnel to create a crisis situation. Shortages of skilled programmers can be partially relieved by careful choice among available programming languages and their compilers. High level languages can make programming easier. Compilers can reduce time spent coding and make up for a shortage of experience by providing the techniques needed to optimize both size and execution speed of machine level code.

What is a compiler?

Software implementation environments can be divided into two levels, as shown in Fig 1: the program machine level and the hardware machine level. Although actual code execution takes place at the hardware machine level, a software engineer cannot efficiently communicate directly with this level. Instead, a programming language, such as PL/M-86, is used as the communication link with the programming machine. The compiler is responsible for translating language input to the programming machine into the language of the hardware machine. In this regard, the maturity of the PL/M-86 compiler as a powerful tool for 8086 software development is revealed.

The compilation process

During the compilation process, the compiler closely binds the input program, determines its syntactic correctness, and generates efficient hardware machine code. Closely binding a program means to fix the types of variables, the forms of the expressions, and the program's structure. To generate efficient hardware machine code, various optimization techniques are used.



The two major steps of the compilation process are the parsing of the input source program and the generation of the output object code. (See Fig 2.) Parsing is achieved by a lexical and syntactic analysis. Lexical analysis separates individual components or tokens making up the program's symbols. These symbols include variable names, key words, and operators. Syntactic analysis checks the program for any syntax errors by determining the structure of the source program in terms of its blocks, statements, and expressions. Results of the parsing are an intermediate text string and a dictionary of variables used in the program.

Generation of the dictionary, or symbol table, is central to the compilation process as it provides a reference for the variable names and their properties. Built during examination of the data declarations, the symbol table is continually referenced during the remainder of the compilation.

The second step of the compilation first performs optimization over the intermediate text, independent of the target hardware. Final object code is then generated, with consideration for hardware machine dependent optimization.

Stanley Mazor is with Intel Corp, 1350 Bordeaux Dr, Sunnyvale, CA 94086, where he has participated in the designs of the MCS-4, MCS-8, 8080, and several other microcomputers. Prior to joining Intel in 1969, he was assistant manager of the computer center at San Francisco State College and a principal designer of the Symbol computer at Fairchild. Mr Mazor has published over 30 articles and papers on microcomputers and shares patents on the 8080 and MCS-4. He is a senior member of the IEEE.

Armond Inselberg is a senior consultant at the Institute for Software Engineering, Suite 200, 535 Middlefield Rd, Menlo Park, CA 94025. He is involved in data processing capacity management for workload analysis and forecasting. Previously, he worked at Intel, Stanford University, and IBM. He has a PhD in computer science from Washington University and an MBA from the University of Santa Clara.

MARCH 1982
ORDER NUMBER: 210397-001

210397-1

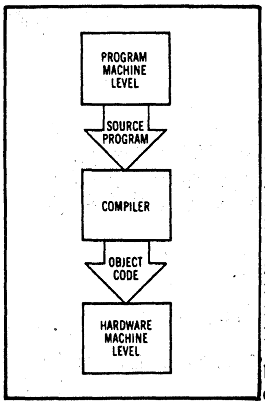


Fig 1 Software implementation environment. Programmer communicates with program machine level, while actual code execution occurs at hardware machine level. Compiler serves as interface between two levels

Optimization philosophy

Efficiency of the generated object code is a primary objective of the compiler. Providing correct code is, of course, the primary objective, but it is never stated explicitly. As with most compilers, the PL/M-86 compiler is geared toward optimizing programs written using good programming practices.

There is a tradeoff between the speed of compilation and the optimization of the resulting object code. Although optimized code is most desirable for finalized production software, the preference during development is for fast compilation. Since a conflict exists between the speed of compilation and code optimization, a compromise must be made.

With PL/M-86, the user can select the level of optimization. Level 0 is the most basic, and level 3, the most advanced. Each successive level provides all optimization techniques of the lower levels, while adding further techniques. If an optimization level is not specified at compile time, the system defaults to level 1.

Specific techniques used within the PL/M-86 compiler serve to optimize the amount of code generated, the execution time of the code, or both the amount of code and the execution time. Hardware machine independent and machine dependent optimization techniques make up a secondary classification of the techniques. Machine independent techniques optimize object code, independent of the target processor. Machine dependent optimization takes advantage of the architecture of the target processor. A third classification is based on whether the techniques optimize over a single program statement or over a range of statements. Table 1 summarizes PL/M-86 optimization techniques for these three classifications.

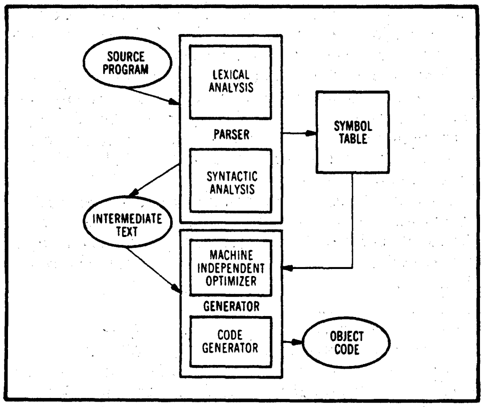


Fig 2 Compilation process. Parsing of source program produces symbol table and intermediate text string. Text string is then optimized, resulting in generation of object code

Amount of code generated

When only a limited amount of memory is available to hold the program, optimizing the amount of code is particularly relevant. Three techniques within the compiler work to reduce the amount of generated code.

Branching to duplicate code—Removing code which occurs more than once, this technique can be used when the paths through duplicate copies of code have the same termination point in the program. In this case, as shown in Fig 3, the second copy of code is replaced with a jump to the original copy.

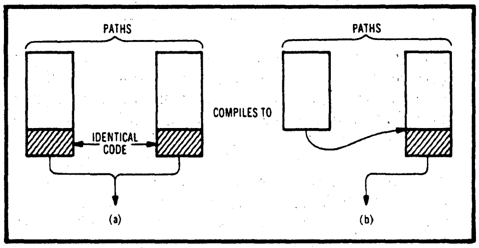


Fig 3 Branching to duplicate code optimization. Both copies of code have same termination point (a); during compilation, second copy of code is replaced by jump to first copy (b)

An example of two program paths that have portions of identical code and terminate at the same point can be found in an IF-THEN-ELSE statement.

```
IF X > Y                                  MOV  AL,Y
   THEN DO;                               CMP  X,AL
        X=Y;        compiles to           JBE  @1
        X=X+1;                            MOV  X,AL
        END;                     @1:      INC  X
   ELSE X=X+1;
```

In the example, the common program statement X = X + 1; is compiled to INC X and is used by both paths through the compiled IF statement. If X is less than or equal to Y, the JBE (jump below or equal) instruction is executed, causing a jump to the INC instruction. If X is greater than Y, INC is reached even though the JBE is not executed.

Removal of unreachable code—This technique causes the compiler to skip those parts of the program that will never be executed. For example, unlabeled program statements that follow a GOTO statement cannot be reached, and therefore will never be executed. Thus,

```
        GOTO LABELZ;
not      IF X > Y         compiles to    JMP    LABELZ
compiled   THEN X=X+1;                   LABELZ: INC Y
        LABELZ: Y=Y+1;
```

210397-2

TABLE 1

PL/M-86 Optimization Techniques

	Amount of Code		Execution Speed		Both Amount of Code and Execution Speed	
	Hardware Independent	Hardware Dependent	Hardware Independent	Hardware Dependent	Hardware Independent	Hardware Dependent
Single statement		Instruction size	Strength reduction		Folding of constants Expression arrangement Short circuit of Boolean expressions Function evaluation	
Range of statements	Branching to duplicate code Removal of unreachable code			Address pointer comparison	Elimination of common subexpressions Elimination of superfluous branches	Peephole Indeterminant storage operations

Although this optimization technique reduces the amount of code generated, it is needed only when the programmer is careless.

Instruction size—The compiler in this case selects the shortest encoding of the instruction. Instructions involving a hardware register can be shortened by one byte if the register is the accumulator. In addition, jumps to locations within 127 bytes require shorter instructions because the increment rather than the target address is specified. For example, if a `JA` conditional jump instruction jumps to a label `@2` that is 14 bytes away, the distance of 14 bytes is stored in the instruction. Thus, the instruction uses one byte to specify an offset rather than four bytes to indicate the target address of `@2`.

```
JA @2          encoded as         77 14
                                  opcode  offset
```

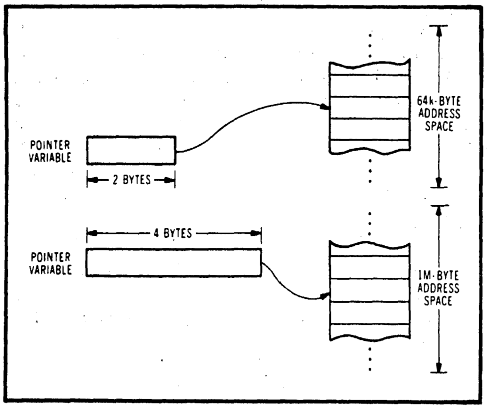


Fig 4 Instruction size optimization. If address space is restricted to 64k (top), compiler allocates 2 bytes for type pointer variable; otherwise, variables require 4 bytes (bottom)

Another aspect of this optimization technique is that the compiler will allocate two bytes to variables declared to be of type pointer, if the address spaces for code and data are restricted to 64k bytes each. Otherwise, as shown in Fig 4, variables of type pointer require four bytes. The programmer indicates the size of the address space to the compiler through a compiler control switch.

Execution speed

Optimizing the execution speed can be critical for time-dependent processing. Two optimization techniques available for improving execution speed are strength reduction and address pointer comparison.

Strength reduction—Execution is optimized by replacing certain operations with faster executing operations. For example, the compiler replaces "multiply a variable `Y` by two" with a shift left operation. The result is the same, but a shift left executes faster than a multiply.

```
X = Y*2;          compiles to          MOV   AL,Y
                                       SHL   AL,1
                                       MOV   X,AL
```

Address pointer comparison—This optimization technique generates code to compare two 32-bit pointer variables. Physical addresses are actually 20 bits, but are stored as a 16-bit base and a 16-bit offset field. When the base is shifted left by 4 bits and added to the offset, it yields a 20-bit address (Fig 5). Execution speed is improved because, instead of calculating the 20-bit address to compare pointers, code is generated to first compare the base parts. Only if the base parts are equal is it necessary to compare the offset parts.

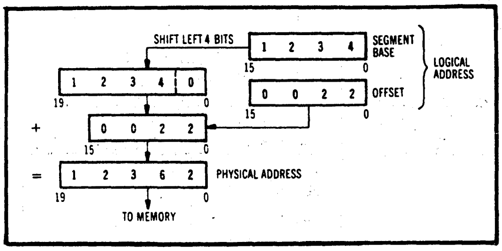


Fig 5 Address pointer comparison. 32-bit pointer variables are stored as 16-bit base and 16-bit offset. Shifting base left 4 bits and adding it to offset results in 20-bit address

For example, two variables, PTR1 and PTR2, are declared to be of type pointer. If PTR1 is greater than PTR2, then X is set equal to 0.

```
DECLARE (PTR1,PTR2) POINTER;
IF PTR1 > PTR2
 THEN X=0;        compiles to       LES   AX,PTR1
                                    PUSH  ES
                                    LES   DX,PTR2
                                    MOV   DI,ES
                                    POP   SI
                                    CMP   SI,DI
                                    JNE   $+4H
                                    CMP   AX,DX
                                    JBE   @1
                                    MOV   X,0H
                         @1:
```

In this example, the LES instruction loads the AX register with the offset of PTR1. The base is loaded into the ES register, then moved to the SI register by means of the stack. The offset of PTR2 is loaded into the DX register and the base is moved to the DI register. The two base values in the SI and DI registers are compared by the CMP instruction. If the results are not equal, the JNE instruction (jump not equal) is executed, skipping the code used to compare the offsets, and jumping to the instruction that sets X to 0.

When only a limited amount of memory is available to hold the program, optimizing the amount of code is particularly relevant.

Optimizing both amount of code and execution speed

Most optimization techniques reduce the amount of generated code and improve execution speed. Eight techniques accomplish this within the PL/M-86 compiler.

Folding of constants—This technique causes the compiler to perform arithmetic operations at compile time rather than at execution time. For example, a statement with the expression 6 + 3 + W would be coded as 9 + W. Thus,

```
V = 6+3+W;        compiles to       MOV   AL,W
                                    ADD   AL,9H
                                    MOV   V,AL
```

Expression arrangement—Code for expression evaluation is generated such that the operations are performed in that order which produces the most efficient code. If expressions I times J and K times L are to be calculated, and their results subtracted, then

```
Z = (I*J) - (K*L);   compiles to   MOV   AL,J
                                   MUL   I
                                   PUSH  AX
                                   MOV   AL,L
                                   MUL   K
                                   POP   CX
                                   SUB   CX,AX
                                   MOV   Z,CL
```

In this example, the result of I * J is pushed onto the stack, freeing the accumulator for a second multiply. After K * L is evaluated, the result of I * J is popped into the CX register. The registers are then subtracted. This process is much more efficient than having the compiler first save the two multiplication results in temporary variables, then move these results to registers, and finally subtract the registers.

Short circuit of Boolean expressions—Generated code terminates the evaluation of a Boolean expression as soon as its outcome is established. For example, consider the expression (V>X AND I>J). If V is not greater than X, the expression will be false, regardless of the results of the rest of the expression; therefore, the remainder of the expression need not be evaluated. Thus,

```
IF (V > X AND I > J)  compiles to   MOV   AL,V
   THEN B=1;                        CMP   AL,X
                                    JBE   @1
                                    MOV   AL,I
                                    CMP   AL,J
                                    JBE   @1
                                    MOV   B,1H
                               @1:
```

In this example, the generated code tests V for greater than X. If this comparison is false, the JBE (jump on below or equal) to label @1 is executed. This label is generated by the compiler to go around the IF statement without executing the remaining code of the Boolean expression. This technique not only saves execution time but reduces the number of generated instructions required to evaluate the expression.

Function evaluation—The compiler evaluates several specific functions as they are encountered in the source program at compile time. For example, for a 10-element array named W, the LAST function obtains the value 9, the last subscript of the array. Arrays are indexed starting with 0.

```
DECLARE W(10) BYTE;
I = LAST (W);                compiles to   MOV   I,9H
```

By evaluating such functions, the compiler saves execution time and storage space, and makes the programmer's job easier by permitting the functions to be referenced.

210397-4

Elimination of common subexpressions—The compiler recognizes multiple occurrences of an expression and saves the value of the expression in a register or stack so that it need not be recalculated. For example, the expression `J + I` or `I + J` may occur several times but will be evaluated only once.

```
X = J + I;          compiles to     MOV   AL,J
Y = I + J;                          ADD   AL,I
                                    MOV   X,AL
                                    MOV   Y,AL
```

By saving the result of `J + I` in the `AL` register, rather than recalculating each time it is encountered, generated object code and execution time are greatly reduced.

Optimizing the execution speed can be critical for time-dependent processing.

Elimination of superfluous branches—Optimization using this technique reduces the number of jumps that must be executed. In the first example, jumping to a `LABELX` that contains a jump to `LABELZ` transforms the first jump into a branch directly to `LABELZ`.

```
IF X > Y                    compiles to   MOV AL,X
   THEN GOTO LABELX;                      CMP AL,Y
                                          JA  LABELZ
      .                                     .
      .                                     .
      .                                     .
LABELX: GOTO LABELZ;                      LABELX: JMP LABELZ
      .                                     .
      .                                     .
      .                                     .
LABELZ:                                   LABELZ:
```

Another example is the selection of a single conditional jump instruction based on the result of a comparison. This optimization can occur frequently, eliminating an unconditional `JMP` instruction each time through the selection of the appropriate conditional jump. Consider the `IF` statement that executes some code only if `X > Y`.

```
IF X > Y           compiles to        MOV  AL,X
   THEN DO;                           CMP  AL,Y
        Z=R;                          JBE  @1
        R=R+1;                        MOV  AL,R
     END;                             MOV  Z,AL
                                      INC  R
                                   @1

                                      MOV  AL,X
                                      CMP  AL,Y
                   compiles to        JA   $+5H
                   without use of     JMP  @1
                   optimization       MOV  AL,R
                   technique          MOV  Z,AL
                                      INC  R
                                   @1:
```

In this example, the `JA` (jump above) and `JMP` (unconditional jump) instructions are replaced by a single `JBE` (jump below or equal) instruction.

Peephole—This optimization attempts to discard redundant instructions. One such action might be loading a register with a value that it contains already. For example, if `Y` is set equal to `X + 1`, the value of `Y` is currently in the accumulators since it was last used to calculate `X + 1`. If `Y` is again used in the next statement, there is no need to fetch the value of `Y`. Thus,

```
Y=X+1;          compiles to       MOV   AL,X
Z=W+Y;                            INC   AL
                                  MOV   Y,AL
                                  ADD   AL,W
                                  MOV   Z,AL
```

Since the value of `Y` is currently in the accumulator as a result of the calculation of `X + 1`, it need not be reloaded into the accumulator for the calculation of `W + Y`.

Indeterminant storage operation—The compiler does not reload the starting point of a based data structure each time that it is referenced. For example, consider `PART` to be an array of structure elements based by the pointer variable `PARTPTR`.

```
DECLARE PART BASED PARTPTR (10)
 STRUCTURE (PARTNO WORD,
            AMT    BYTE,
            COST   WORD);

PART(2).PARTNO=6C4H;  compiles to   MOV   BX,PARTPTR
PART(6).AMT=79H;                    MOV   PART[BX,0AH],6C4H
                                    MOV   [BX+20H],79H
```

The first reference to the array structure places the base of the array, contained in `PARTPTR`, in the `BX` register. Further references to the array structure do not require that the `BX` register be reloaded.

Evaluation examples

PL/M-86 offers four levels of optimization. Optimization techniques provided at each of these levels are classified in Table 2. To indicate how much storage is actually

TABLE 2

Optimization Techniques Provided In Each Compiler Level

Optimization Technique	Optimization Level			
	0	1	2	3
Folding of constants	X	X	X	X
Expression arrangement	X	X	X	X
Short circuit of Boolean expression	X	X	X	X
Function evaluation	X	X	X	X
Strength reduction		X	X	X
Elimination of common subexpressions		X	X	X
Elimination of superfluous branches			X	X
Removal of unreachable code			X	X
Branching to duplicate code			X	X
Instruction size			X	X
Peephole			X	X
Indeterminant storage operations				X
Address pointer comparisons				X

210397-5

TABLE 3

Object Code (bytes) Generated For Each Optimization Level

	Level 0	Level 1	Level 2	Level 3
Program A: Mastermind	1688	1559	1450	1450
Program B: General sort	1953	1789	1503	1503
Program C: Frequency count	849	765	694	694
Program D: Process simulation	7955	7951	7083	7083
Program E: Service queue	289	250	212	185
Average % size reduction from previous level		7.9%	12.28%	2.55%

saved by these techniques, five sample programs were compiled at each level using version 2.1 of the compiler; Table 3 provides the size in bytes of resulting compilations. The reduction in size obtained in going from one level to the next higher level is due to the additional optimization techniques used at the higher level.

Programs used in this study demonstrate the compiler's ability to optimize various types of instructions. Program A plays the game of mastermind with the operator performing a large amount of input/output with the cathode ray tube. Program B performs a sort on an array of 1000 records, making extensive use of structures and pointers. Performing a frequency word count on an arbitrary text file, Program C uses string move instructions and pointers. Program D uses simple coding with no structures or pointer addressing to perform a process simulation. Service queue simulation using linked data structures is done in Program E.

For each successive level of optimization, the individual percentages in size reduction of the programs were averaged. From Table 3, it becomes apparent that Level 3 optimization provides nearly a 25% reduction in storage requirements.

Conclusion

As the demand for microprocessor software increases, the selection of the implementation language will receive more attention. In choosing a language, users must consider not only high level constructs of the language itself, but also the capabilities of available compilers to translate the resulting programs.

210397-6

intel®

ARTICLE
REPRINT

AR-239

November 1983

PL/M-51: A High-Level Language for the 8051 Microcontroller Family

Rajen Jaswa
Wescon/82

Originally prepared for and presented at Wescon/82

ORDER NUMBER:210806-001

210836-1

PL/M-51: A HIGH-LEVEL LANGUAGE FOR THE 8051 MICROCONTROLLER FAMILY

High-level language advantages are fairly well recognized now. Developing software for embedded microcontrollers using assembly language is labor intensive and therefore an expensive task. It is not easy to come up with a sequence of well-defined stages to go from the system design stage to the system implementation software. The transformation of an algorithm flowchart to the actual assembly-language code requires considerable intuitive guesses and inventiveness on the part of the programmer. Also, assembly language is difficult to read and inspect. Because assembly language projects are difficult to manage, there has been a widespread movement towards using high-level languages. High-level languages provide, in general, improved programmer productivity, and reliable, maintainable, portable software.

In the microcontroller environment, the major considerations for a high-level language are efficient code, close control over hardware resources and optimum use of scarce on-chip data memory (RAM is very expensive in terms of silicon real estate). Intel developed PL/M-51 for the 8051 single-chip microcontrollers with the specific goal of trying to meet these criteria with minimal impact on the traditional high-level language benefits of reliability and maintainability.

OVERVIEW OF THE 8051 ARCHITECTURE

The 8051 is a stand-alone high-performance single-chip computer intended for use in sophisticated real-time applications such as instrumentation, industrial control and intelligent computer peripherals. It provides the hardware features, architectural enhancements and new instructions that make it a powerful and cost effective controller for applications requiring up to 64K-bytes of program memory and/or up to 64K-bytes of data storage. Figure 1 shows the 8051 Functional Block Diagram.

The 8051 microcomputer integrates on a single chip the CPU, 4K x 8 read-only program memory, 128 x 8 read/write data memory, 32 I/O lines, two 16-bit timer/event counters, a five-source, two-priority level, nested interrupt structure, serial I/O port for either multi-processor communications, I/O expansion, or full duplex UART, and on-chip oscillator and clock circuits.

The 8051 has four address spaces tailored to support a wide range of control applications efficiently—program memory, on-chip and external data memory, and the bit memory space. This complex (but sophisticated) memory architecture is supported by a rich (but unorthogonal) set of addressing modes for efficient memory access—register addressing, direct, indirect, immediate and base-register plus index-register indirect addressing. To support this complex memory architecture, a high-level language's syntax must mirror the underlying microcontroller architecture. The challenge is to implement this without compromising the language's readability and maintainability.

The popular 8051 architecture forms the core of the MCS-51™ microcontroller family. The need to base processors on a popular, industry-standard architecture is dictated by the cost of developing processor support hardware and software tools, as well as a desire to maintain the customer's investment in engineering resources and capital equipment. The upgradeability requirement has to be traded off against providing optimum functionality in the processor for the target market segment. Consequently, the 8051 family consists of straight-line enhancements—RAM, ROM memories and clock rates—as well as microcontrollers like the 8044 remote universal peripheral interface processor (RUPI), which has the 8051 core architecture but supports an interrupt structure and I/O functions tailored to the distributed processing environment. The cost of developing a new support environment for processors targeted to specific (and small) market niches would make the processor an unviable product. Consequently, software tools for proliferation processors should be configurable from the core processor support products.

210836-2

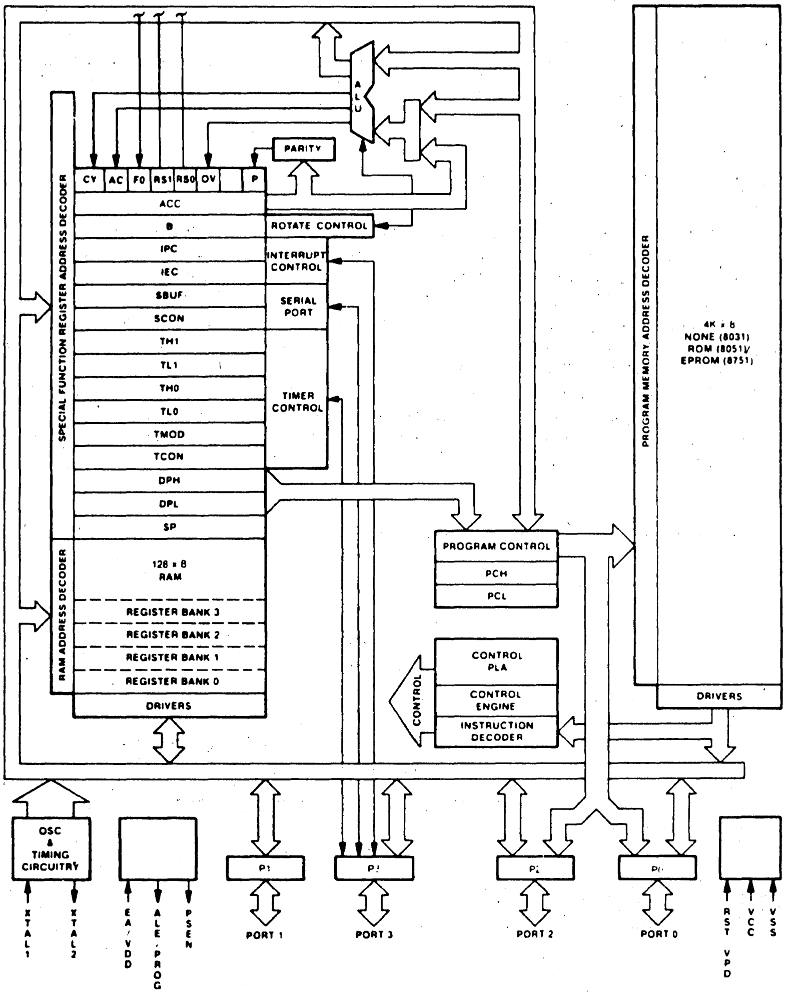


Figure 1. 8051 Functional Block Diagram
Copyright INTEL CORPORATION, 1981

210836-3

PL/M-51

PL/M-51 was developed to facilitate the design of reliable, maintainable microcontroller systems. This goal translates into a programming language which encourages and enforces good software engineering practices such as structured programming, top-down design and implementation, step-wise refinement and software walk-throughs. However, this goal has to be traded off against the exigencies of the microcontroller environment—high performance requirements, scarce memory resources and control over the hardware facilities. PL/M-51 tries to satisfy these conflicting requirements by enforcing block structured software design, providing control-flow statements for structures programming (if-then-else, do case, do while, . . .) as well as by supporting 8051 architecture specific attributes at the language level, for example—the REGISTER and AUXILIARY variable attributes, and the specifics of interrupt handling.

SOFTWARE ORGANIZATION WITH PL/M-51

Most applications are decomposed into logically related functions which can be programmed more or less independently of other functions. Interactions between functions are via a few well-defined data parameters and system level status blocks which are globally accessible to all functions at all times. PL/M-51 program structure maps very well into this structured software organization. PL/M-51 programs consist of one "main" module and several functional modules which are independently compilable units and consequently can be independently developed and debugged. Each module consists of one or more procedures. A procedure contains variable declarations and a sequence of executable statements. Variables have restricted scope to the block they are defined in, unless the scope has been extended by the PUBLIC/EXTERNAL attribute. The advantage of block scoping of variables is that programming errors of duplicate variable use are quickly identified. Figures 2 and 3 show the organization of PL/M-51 programs for heirarchical tree-structured real-time software systems. PL/M-51 does not enforce a tree-structured organization, but it provides a modular organization facility for implementing it.

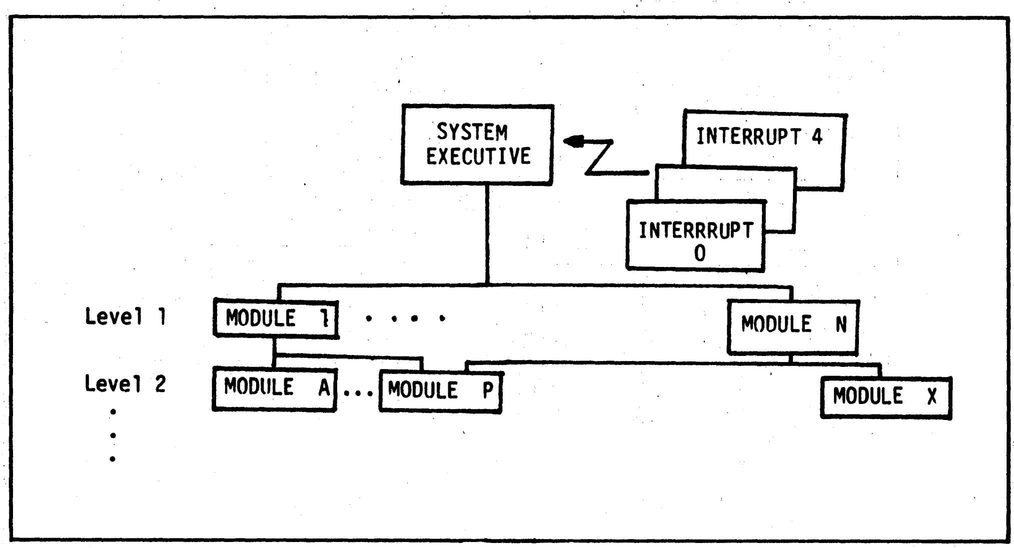
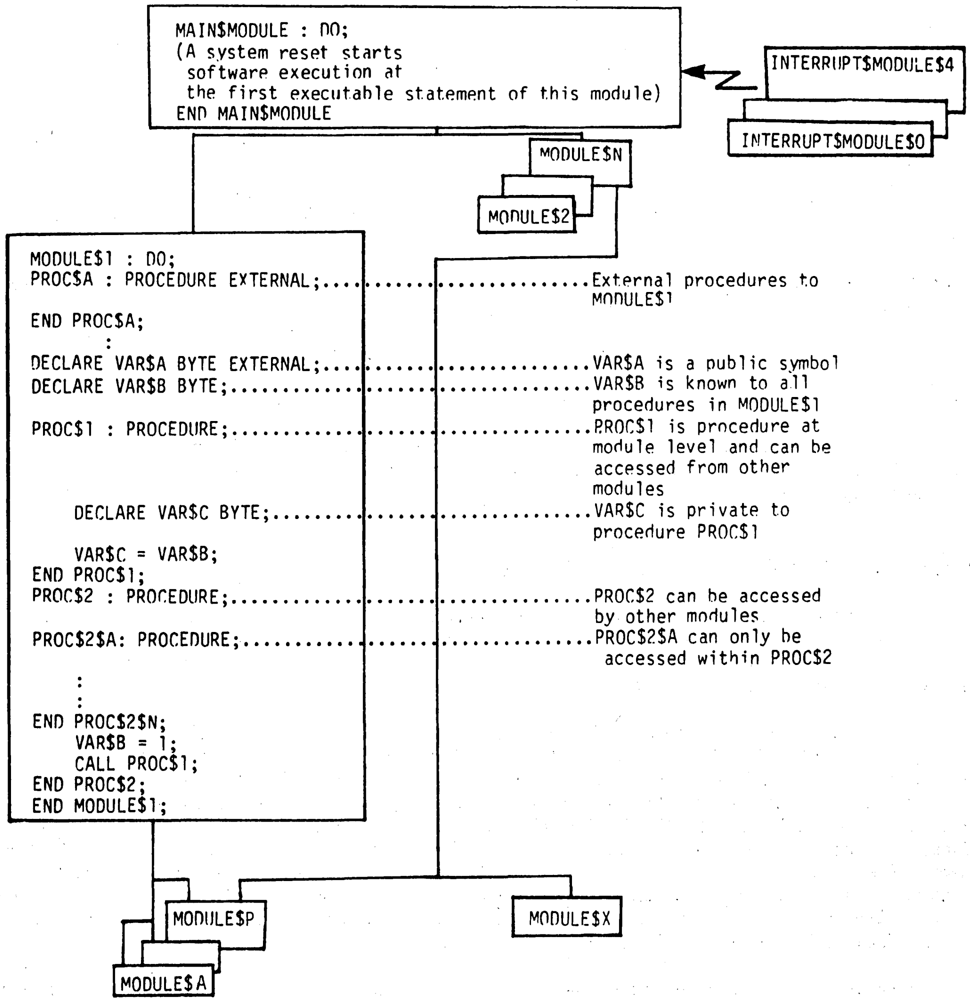


Figure 2. Hierarchical Real-time Software Systems

210836-4



```
MAIN$MODULE : DO;
(A system reset starts
 software execution at
 the first executable statement of this module)
END MAIN$MODULE
```

```
MODULE$1 : DO;
PROC$A : PROCEDURE EXTERNAL;.........................External procedures to
                                                      MODULE$1
END PROC$A;
         :
DECLARE VAR$A BYTE EXTERNAL;.........................VAR$A is a public symbol
DECLARE VAR$B BYTE;..................................VAR$B is known to all
                                                      procedures in MODULE$1
PROC$1 : PROCEDURE;..................................PROC$1 is procedure at
                                                      module level and can be
                                                      accessed from other
                                                      modules
    DECLARE VAR$C BYTE;..............................VAR$C is private to
                                                      procedure PROC$1
    VAR$C = VAR$B;
END PROC$1;
PROC$2 : PROCEDURE;..................................PROC$2 can be accessed
                                                      by other modules
PROC$2$A: PROCEDURE;.................................PROC$2$A can only be
                                                       accessed within PROC$2
    :
    :
END PROC$2$N;
    VAR$B = 1;
    CALL PROC$1;
END PROC$2;
END MODULE$1;
```

Figure 3. Organization of PL/M-51 Programs

210836-5

DATA TYPES

8051 microcontroller software requires intimate knowledge of the machine representation of data variables because a significant amount of processing is done at the bit level. Consequently, the basic types of data in PL/M-51 are BIT, BYTE and WORD—as opposed to INTEGER, REAL . . . COMPLEX machine-independent data types in other high-level languages. With the three basic data types of PL/M-51, the state of each variable is known to the programmer—at the bit level. This is important, if PL/M-51 programs are to take advantage of the powerful boolean instructions on the 8051.

BUILT-IN FUNCTIONS

The PL/M-51 language has been enhanced with a number of useful standard functions which provide information about data representation at run-time to programs, do type conversions and provide machine level functions at a high-level language.

The LENGTH and index of the LAST element in an array and the SIZE of a variable in bytes can be obtained by a program at run-time. This facility permits the development of program libraries which can be reused on other projects.

System programs require the ability to manipulate data at the machine representation level as well as at the logical level. Consequently, PL/M-51 provides type conversions BIT to BYTE to WORD as well as machine level instructions like rotate and shift for variable manipulation.

The 8051 architecture has a powerful instruction repertoire for conditional execution on bit states. PL/M-51 provides a TESTCLEAR function to support process synchronization primitives—for example, semaphores require uninterruptible test-set atomic operations.

8051 ARCHITECTURE SPECIFIC ATTRIBUTES

The 8051 architecture is designed to provide optimum performance over a wide range of control applications. Consequently, it has a sophisticated (and complex) memory organization, and four register banks in the central processing unit (CPU) for rapid task switching during interrupts. PL/M-51 supports programming for this environment by embracing architecture specific attributes within the language syntax.

Memory mapping of variables is done by specifying a suffix attribute during data declaration. The possible attributes are CONSTANT, AUXILIARY, REGISTER AT (128-255), MAIN and IDATA. CONSTANT variables reside within the code memory, while AUXILIARY variables are assigned to off-chip data memory. The default memory assignment or MAIN variables reside within the directly-addressable on-chip data memory. IDATA variables are indirectly-addressable over the entire on-chip data memory (0–255). The REGISTER attribute maps the variable to the pre-defined mapped registers, I/O ports and functions on-chip. The compiler generates the appropriate addressing instructions to access these variables. The key benefit of letting the compiler generate addresses (mechanically) is that when decisions to move variables from one memory space to another are made, only the declaration attribute has to be modified, and the module recompiled. The impact of such an action is an assembly language program would require identifying all references to the affected variable and changes in its code an error-prone and laborious job.

Rapid response to events are key to high performance in control applications. The 8051 architecture provides four register banks and task-switching requires only the program counter, program status word, A, B and DPTR registers to be saved. PL/M-51 allows procedures to be associated with a particular register bank. Only the program counter, not the RO-R7 register bank, needs to be saved on the stack during a subroutine call, since they use the same register bank. Task switching and the associated register bank switching is supported by the interrupt mechanism for external and internal events.

Interrupt service routines are identified by associating the hardware INTERRUPT number attribute to a procedure. The register bank too should be identified for the interrupt service routine. To prevent data corruption, interrupt service routines should use different register banks than non-interrupt code. Also, low and high priority interrupts should not use the same register bank. Since it is illegal to call procedures using different register banks, communication of information from interrupt events have to be handled via shared global data areas.

210836–6

A GENERIC COMPILER

The rapid development of silicon technology allows semiconductor houses to optimize processors to specific market segments. For example, the 8044 slave processors provide intelligent peripheral control and are based on the 8051 CPU architecture. PL/M-51 can be configured to support the 8044 by inputting to the compiler a processor definition file which has information about register names and memory mapping of I/O functions and bits. Configurable compilers provide an optimum approach to managing the costs of maintaining system software, as well as supporting proliferation processors based on successful CPU architectures.

CONCLUSIONS

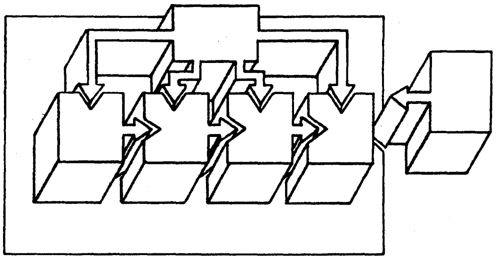
Software development for microcontroller applications can be executed in a planned methodical manner. PL/M-51 provides software engineers with a tool for promoting structured software design for the 8051 microcontroller family. PL/M-51 provides an environment for controlled system development.

210836-7

INTEGRATED TOOLS ACCELERATE CODE DEVELOPMENT

Integrated source and version control, electronic mail, and standard interfaces for programming languages and operating systems can move the software task faster than using additional programmers.

by Dennis Carter



If a project is running behind schedule, adding staff members is not always the best tactic for getting it back on schedule. As the saying goes, adding manpower to a late software project makes it later. Often the best solution is to coordinate programming efforts and project management through an integrated development environment. This type of system stimulates greater efficiency by combining management, programming, and debugging tools in one environment. Productivity increases especially for microprocessor systems with separate target and host development systems. As a result, industries can meet critical delivery schedules without needing additional programmers.

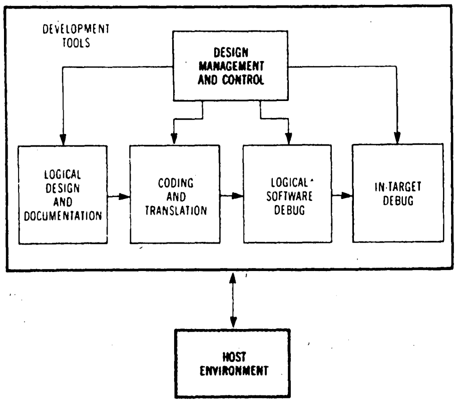
System development is a complex process involving several different stages that continually pass information between each other. The development environment should be more than a collection of assorted tools that are poorly linked. It must efficiently coordinate the diverse stages of development in a single environment, allowing information to flow easily between different tiers of the project.

An efficient development cycle has two parts. Managers must have a clear view of the project from inception through test and implementation. Thus, planning work schedules and anticipating design bottlenecks are easier. Software engineers must share their ideas, designs, and programs—passing information throughout the different development stages.

Yet, in developing products for other target machines, an integrated environment for the host development system alone is not enough. Unless a smooth transition to the final target environment

Dennis Carter is software product marketing manager for Intel (Santa Clara, Calif). He holds an MBA from Harvard University and an MS in electrical engineering from Purdue University.

Reprinted with permission from *Computer Design*, January 1985 issue.



An integrated development environment must do more than act as a library for development tools. It must ensure that information flows smoothly between components. As organizations shift to new development policies and expand development hardware, the system must be able to migrate smoothly to the new host environment.

is provided, the project will bog down during the critical target system integration and test. The transition from host to target development environments is one of the two major factors affecting the project cost. According to Randall W. Jensen, chief scientist at Hughes Aircraft Co, changing environments can increase costs as much as 122 percent.

Host hardware environments also change as the company expands its development resources. Rather than losing previous investments in tools or training, the company must be able to shift the entire environment smoothly. Some workstations are built to make this transition easy. For example, engineers using Intel's Intellec® Series IV workstation maintain the same fundamental development environment when they move to the NDS-II distributed development environment.

With its multiple stages, system development can turn into a logistical headache for managers and engineers alike. Managers supervising several programming teams, each developing different versions of programs, can easily lose the thread of revisions to the source code. Similarly, programmers can find themselves working at cross-purposes in their attempts to generate and test the most recent versions of code, rather than a hybrid of current and obsolete code versions.

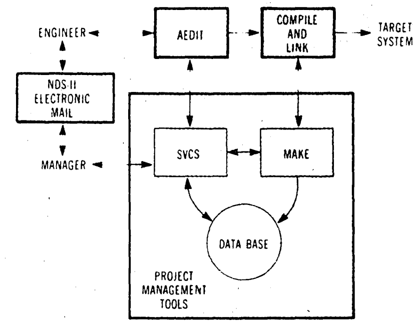
An integrated system can help prevent these problems by combining different tools and making them work well together. For example, Intel's configuration management tools, Source Version Control System (SVCS) and MAKE, manage multiple versions of a program. The tools can automatically combine the most current versions of several modules in larger programs. Similarly, Intel's debugging aids, PSCOPE and Integrated Instrumentation and In-Circuit Emulation (I^{2}ICE™) package, use information implanted by compilers to permit programmers to debug during the integration process at the source level. Such an integrated environment increases efficiency through good allocation of available resources.

Management and control

Modular design helps software engineers break a large complex problem into a set of small simple programs. Unfortunately, a modular design system requires more overhead for managing a large number of modules and different versions of the same module. If the logistics become too troublesome, programmers might even collapse several modules into a single file to save themselves the trouble of manipulating the separate modules. Project management tools can free engineers from the housekeeping chores associated with program development.

Programmers keep track of major changes in their programs by either creating copies of the new version or changing an older version. The result is a series of similar programs that lack proper documentation to indicate the change and reason for the change. SVCS provides an automated approach to this record keeping. It tracks changes to the baseline version of a program, and demands that programmers record their reasons.

When software engineers need a particular version of a file, whether the current or some older copy, SVCS automatically retrieves the correct version from its data base of updates and baseline versions. Similarly, after the programmers have added changes, SVCS records the updates and the reasons for the



Besides controlling changes to the source files in its data base, SVCS helps managers audit source updates. Automatically generating the software for the target system, MAKE reduces generation time by about 50 percent, leaving engineers more time to concentrate on development.

231433-2

changes, adding as little as a 3 percent overhead. In addition, SVCS helps project managers exercise precise control in large team projects by preventing certain engineers from making changes independently.

While programmers work directly with SVCS to manage different program versions, MAKE works closely with SVCS facilities to generate current versions of systems. While generating large systems from several different modules, programmers often find that one or two modules have been updated since the last compilation. This problem is compounded when modules depend on a series of other submodules. MAKE automates the manual procedures often resorted to by software engineers to track current object modules.

Management tools can free engineers from housekeeping chores.

Using templates that detail the modules' interdependence, MAKE ensures that only current versions of modules are included in the system generation. If it finds that a required object module is obsolete, MAKE will automatically compile the appropriate source module to produce the current version of the object module. Furthermore, if source modules depend on submodules, MAKE will continue searching through its templates to ensure it recompiles modules using the current submodules for these source modules.

MAKE selectively compiles the needed modules. Only if a module or one of its submodules is obsolete does MAKE execute a recompilation. This cuts the inefficient massive compilation procedures commonly used to ensure that object modules are current.

In addition to the project management tools handling version control and system generation, a complete integrated development environment should also facilitate communication among users. Acting as an electronic central distribution center, the NDS-II electronic mail facility maintains mailboxes for individual users and groups of users on the network, and an electronic bulletin board for all users. In addition to supporting document distribution, electronic mail manages a file transfer facility. Team members can transmit both source and object modules to any other user on the network.

Another feature, NDS-II's network resources manager (NRM), provides extensive support for file management and resource sharing. The NRM manages files with a hierarchical structure that arranges files into volumes and multiple subdirectories. The NRM also improves allocation of resources through its distributed job control (DJC) facility. DJC permits users on private workstations to export a batch job to the NRM for remote execution. The NRM then moves the job to a free workstation for execution, returning the completed job status to the user's directory.

Logical design

An integral part of the software development environment and its primary interface with the user is the text editor. Because software engineers typically spend 40-50 percent of their time using a system editor, it is a critical element in software development and can greatly enhance productivity if used well. For example, programmers often need to work simultaneously on two separate files, such as two different source programs or a program and a specification document. Editors such as Intel's AEDIT permit them to edit two files of any size simultaneously and transfer text between them.

AEDIT's ability to store a sequence of edit commands also simplifies the use of edit macros. With AEDIT, programmers build macros simply by typing in their commands. They can reexecute the command series or save it on disk for later use. AEDIT also helps software engineers with structured programming techniques through its automatic text indentation. Furthermore, AEDIT protects programmers' efforts by optionally creating backup copies of files being edited.

Although a text editor serves as the primary interface between the development system and programmer, programming languages serve as the principal interface between design concepts and the target hardware. With the right set of programming languages and support tools, software professionals can develop the optimal solution for a particular situation, without the design bias often seen when designers plan projects with an eye on their eventual implementation.

For example, different programming languages like assembler, PL/M, C, Pascal, and Fortran enjoy certain advantages over each other. Software developers should be able to draw on the most appropriate language to implement the different facets of a design. In order to support this kind of free choice, however, the development environment must be able to coordinate the use of a mix of programming languages, so that programmers can use different languages without concern about how the different modules will eventually be combined.

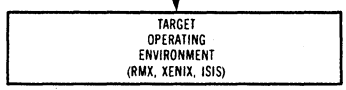
Like spoken languages, the virtue of programming languages lies in their ability to represent abstract ideas in concrete terms. Just as it may be easier to express a certain idea with a particular spoken language than another, programming languages vary in their ability to represent certain design concepts. For example, software engineers find that Pascal represents structured designs more faithfully than a language like Fortran. Also, languages like PL/M or C, which closely reflect the hardware base of a design; or assembly language, which provides the ultimate visibility into the hardware, are powerful tools for developing realtime embedded systems.

Still, programming languages share another feature with natural languages—varying degrees of popularity. For example, Fortran remains one of the most popular programming languages. Its continued strong momentum translates into a large installed base of software. For managers, this large installed base provides a ready source of existing code. On the other hand, managers must remain ready to incorporate newer languages like Ada into designs without starting from scratch.

In many software development projects, managers often look for a way to juggle several programming languages simultaneously. Software engineers can usually adapt quickly to new programming languages—particularly when they are supported by project management tools. On the other hand, the development environment often acts as a bottleneck in mixing several different languages in the same target system because of its inability to match the varying program and system interfaces of different languages.

The Intel development environment integrates different languages through a common object module format (OMF). A standard OMF works at several levels. During link time, OMF presents a standard method for indicating data type information, which the linker uses to build its memory allocation tables. Furthermore, debuggers exploit OMF's standard arrangement of symbolic information for handling symbolic debugging.

Two other aspects of the standard development environment include the definition of standard conventions for passing parameters between different programs—regardless of their implementation language—and standard interfaces to the operating environment. Besides accounting for critical implementation details another key measure of the effectiveness of a development environment is its support of application level standards like IEEE 754 for floating point operations or IEEE 802 for Ethernet.


APPLICATION PROGRAM
CALL DQ$ALLOCATE
UNIVERSAL DEVELOPMENT ENVIRONMENT
ALLOCATE MEMORY
TARGET OPERATING ENVIRONMENT (RMX, XENIX, ISIS)

Where application standards do not already exist, a development system should follow some baseline. The universal development interface sets a baseline for interactions between application programs and operating software. For example, an application that requires memory uses a UDI call (DQ$ALLOCATE), which is later translated into the appropriate call for the target operating environment.

System-independent interface

For those areas currently without standards, the development environment takes the initiative with a baseline for the operating environment. Here, Intel's universal development interface (UDI) defines a system-independent interface between application programs and the operating environment. Rather than write their programs with system-dependent calls to operating system utilities, software developers use the same UDI call to allocate memory, for example, regardless of the target operating system. During link-time, the linker uses this UDI call to link in the appropriate system utility in RMX, for example. Consequently, programs that use the UDI can be ported between ISIS, RMX, and Microsoft's Xenix simply by loading the modules into the new environment. Thus, if the design calls for a realtime operating environment like RMX, engineers can develop the application under ISIS without fear that their work will be lost when the system is transported to the RMX environment.

For the manager trying to improve productivity, no faster method exists than simply porting existing code to a new environment. Besides IEEE standards, which provide a common application environment, the use of a OMF and UDI provide a clear migration path between different operating environments.

In the kind of cross-development environments commonly used for creating microprocessor-based products, engineers work most effectively if they are able to split debugging into two phases. In the first phase, debugging occurs in parallel for the target hardware system and for the software. Here, engineers use the host environment to debug the basic logic of the software system. Once they are satisfied both with the logic of the software and with the operation of the hardware, the engineers then load the software into the target system for the second phase—integration and test.

This in-target phase is the critical step where hardware and software are finally integrated as a total system. As noted earlier, differences between the host and target environments can more than double costs. Consequently, a key feature of an integrated environment is a common debug interface between host and target.

Intel's PSCOPE debugger permits programmers to check out programs at the source level both during logic debug and during in-target test. Because

231433–4

Debugger Command Language

Command	Action
Go/Listed/Pstep	Control program execution
Define Display Modify Remove	Manipulate debugger or program objects
Call/Return	Execute debugger procedures
Call/Return	Execute debugger procedures
Write/CI	Console input/output
Do/End	Define command blocks
Repeat/Count	Repetition of commands or command blocks
If/Then/Else	Conditional execution of commands or blocks
Input/Put/Append	Save/restore to and from disk

PSCOPE shows up again as one of the three major components of the I^{2}ICE system, software engineers are assured of a smooth transition between host and target. Along with PSCOPE, the I^{2}ICE and the logic timing analyzer (LTA) give developers a full view simultaneously into the hardware and software components of their systems. Without this kind of coordinated approach to system integration and test, developers can never deal with the hardware and software as an integrated system, but are forced to switch continually between hardware testing and software debugging.

Supporting system integration at the most fundamental level, in-circuit emulation provides a transparent, full speed emulation of the iAPX 86 and iAPX 286 families of processors. Besides handling multiple level breakpoints and traces in single microprocessors, I^{2}ICE extends its support to multiprocessor environments. Developers can emulate a system of up to four microprocessors and examine complex processor interactions like synchronization. For example, I^{2}ICE lets engineers define events like breaks and traces conditionally, so that a microprocessor will break when another defined event occurs in a different microprocessor.

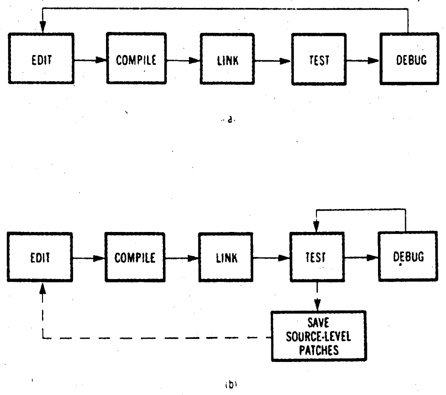
While I^{2}ICE and PSCOPE provide the fundamental support for a system's underlying hardware and software, the LTA also serves as a key element of the system's integrated package. Displaying 16 channels of logic and timing information, the LTA helps isolate critical state and timing problems. In order to speed the analysis process, this menu-oriented system also permits engineers to save debugging setups and waveforms on disk.

A key advantage of an integrated environment is its ability to present information, through a consistent command language, in a familiar form. With I^{2}ICE, this feature extends to logic and timing analysis. Rather than present a morass of digits, the LTA displays most information in easy to understand waveform diagrams.

Source-level debugging

Just as the LTA has moved system integration and test above the bit level, PSCOPE shortens software debugging by permitting engineers to test programs using their own symbols, rather than machine code. With the traditional machine code debugger, if they wanted to patch a section of machine code, programmers would spend hours converting machine code between different formats, like binary and hex, and calculating the machine code equivalents of assembler instructions. Even somewhat more sophisticated debuggers that disassemble machine code are little help in retaining the sense of a program as expressed through its use of symbols.

Instead, even though it helps software engineers deal with machine code when necessary, PSCOPE can handle debugging at the level of the original source code. Consequently, programmers can set an unlimited number of breakpoints by statement number, step through a single source statement at a time, and trace execution by statement number,



In the past, engineers have needed to iterate through a lengthy development cycle in order to debug source code in the target system (a). On the other hand, PSCOPE lets engineers use source-level code to debug and patch the target system and continue debugging. Then, after many bugs are found, PSCOPE saves the source-level patches on disk for later addition to the original source files (b).

231433-5

procedure name, or label (regardless of whether they are working with the host or target system).

From the user's point of view, the utility of PSCOPE lies in its built-in, CRT-oriented editor and in its command language that resembles a high level structured programming language. Using PSCOPE's editor, engineers can write extensive procedures in the command language for testing code and even for patching existing code with new or revised source statements.

The many advantages of an integrated environment include source-level debugging tools, such as PSCOPE.

PSCOPE's ability to handle source-level patches avoids the conventional development scenario where software developers go through a continual cycle of edit-compile-link-test-debug. Source-level patching short-circuits this loop; programmers can remain in the debug phase—patching at the source-level and even saving the source-level patch on disk for later incorporation into the original source-code files maintained under SVCS.

The advantages of an integrated environment show up here dramatically. During compilation, the compiler places symbolic information associated with a program into the object modules it generates. In turn, the linker carries this information along into the runtime image. Both PSCOPE and I^{2}ICE draw on this symbolic information for their source-level debugging. Consequently, during system debugging, developers see familiar procedure and data names, rather than a confusing series of machine codes or disassembled mnemonics. Furthermore, because it maintains this symbolic information in a virtual table, PSCOPE is able to handle arbitrarily long symbol tables—it just brings a new page of symbols from disk, if necessary.

As a result of its ability to coordinate its tools for the various stages of development, the Intel development environment lets system engineers concentrate on product development, rather than on administrative chores. For the development manager, this translates into on-time product delivery, without the costs of additional resources.

231433–6

Ada Task Synchronization in a Multiprocessor System with Shared Memory

Timothy E. Lindquist and Richard C. Joyce
Department of Computer Science, Virginia Tech, Blacksburg, VA 24061

Ada provides a means for concurrent processing within a program through tasking. Several asynchronously executing tasks may constitute a single Ada program. Intertask communication and synchronization is provided through a rendezvous mechanism. This article presents an implementation of Ada tasking for a multiprocessor system having a shared memory. The INTEL 80286 processor is used as an example basis for such a system. The code needed to implement synchronization and communication among tasks (rendezvous) executing on possibly distinct processors is presented. Intertask message passing is an important aspect of the applications for which Ada will be used. This article addresses efficient message transmittal in the context of shared memory.

INTRODUCTION

The implications of using the Ada programming language to produce software for a strict real-time environment are largely unknown. This lack of experience is compounded when the target system has distributed or multiprocessing characteristics. Since Ada will be the common high-order language for use in future Department of Defense embedded systems, efficient implementations of Ada facilities on architectures critical to embedded applications are important.

Architectures representing increased difficulty of Ada implementations include single-processor systems, multiprocessor systems with shared memory, and multiprocessor systems without shared memory. When matching these variations with the possible levels of distribution of an Ada program, several combinations of systems exist. The distribution of an Ada program may range from (1) no distribution—a single Ada main program together with all its tasks run on a single processor, (2) fixed assignment of program parts to processors—for example, on program initiation all tasks are assigned a processor and that assignment remains throughout program execution, (3) dynamic assignment of program parts to processors—in this case the assignment of a program part to a processor may change during execution; for example, tasks may migrate among processors.

For fixed and dynamic assignment, the Ada object of distribution need not be the task. Based on the programmer's view of the application, assignment may be done for packages or data structures. For example, Cornhill [1] is considering program distribution based on programmer-defined names. Others, such as Roberts [5], have examined the Ada task

Journal of Pascal, Ada, & Modula-2, Vol. 4, No. 1, pp. 9-19 (1985)
©1985 by John Wiley & Sons, Inc. Reprinted with permission.

CCC 0735-1232/85/010009-11$04.00

231543-1

as the object of distribution. In this article we examine efficient synchronization of tasks assuming there is a fixed assignment of tasks to processors. Our approach builds upon the ideas presented by Habermann [2], in which scheduling points are minimized for task synchronization. While Habermann's solution is tuned to message-passing applications of Ada in which replies are not expected, our solution is not tuned to a specific application of tasking. We assume a multiprocessor architecture based on the INTEL 80286, with a shared memory. The shared memory allows techniques used in single-processor systems to be adapted, but presents efficiency problems in synchronizing and communicating among processors.

ADA TASKING

An Ada program is made up of one or more tasks each of which executes on its own logical processor. Tasks are executed asynchronously except at points of programmer-specified synchronization (and communication). A task is declared and initiated in an Ada program by (1) declaring a task type, (2) detailing a task body for the type, (3) creating an object of the task type.

A task type is declared through a task specification, which includes entries for the type. Entries may be parameterized and called by other tasks in much the same way as procedures are called. Below is an example task specification for the task type BUFFER with two entry points SEND and RECEIVE. In this example, a BUFFER task synchronizes transmittal of messages from a sending task to a receiving task.

```
task type BUFFER is
  entry SEND (CH : in MESSAGE);
  entry RECEIVE (CH : out MESSAGE);
end BUFFER;
```

A task body is the section of code that is associated with a task type. The body details the actions that are to take place within instances of the task. Accept statements are placed in the body to correspond with entries. A rendezvous occurs between two tasks when one executes an entry call and another executes a corresponding accept statement. The caller does not continue until the accept has completed. The calling and called tasks stay synchronized until the accept completes, after which each continues asynchronously. Below is a task body for BUFFER in which messages are obtained from the sending task and relayed to the receiving task one at a time (buffer_size = 1).

```
task body BUFFER is
  MSG : MESSAGE;
  begin
    loop
      accept SEND (CH : in MESSAGE);
        MSG := CH;
      end SEND;
      accept RECEIVE (CH : out MESSAGE);
        CH := MSG;
      end RECEIVE;
    end loop;
end BUFFER;
```

Figure 1 demonstrates one possible use of the BUFFER task shown above. The MSG_HANDLER task buffers a single message, the PRODUCER task places messages in the buffer using SEND, and the CONSUMER procedure, which runs as a separate task, reads messages from the buffer using RECEIVE.

A task body may contain more than one accept statement for an entry, but all accept statements within a task must be for its own entries. If an entry is called before a corresponding accept statement is encountered, the call is queued for the entry. When an accept statement is executed before any entry call is made, the accepting task suspends until a task calls the entry.

Consider the following example execution of Figure 1 that demonstrates an accept before an entry call. CONSUMER is invoked, and the task objects

```
procedure CONSUMER is

  MSG_HANDLER : BUFFER;

  task PRODUCER;
  task body PRODUCER is
    MSG : MESSAGE;
    begin
      loop
        -- build a message in MSG
        MSG.HANDLER.SEND(MSG);
        -- exit when no more messages
      end loop;
  end PRODUCER;

  MSG : MESSAGE;

begin
  loop
    MSG_HANDLER.RECEIVE(MSG);
    -- use the message in MSG
    -- exit when no more messages
  end loop;
end CONSUMER;
```

Figure 1. An example of Ada tasking.

MSG_HANDLER and PRODUCER are elaborated. These tasks are activated before execution of the first statement of procedure CONSUMER. Assuming that PRODUCER has not yet executed a call to SEND, when MSG_HANDLER executes the accept statement for SEND it will suspend. When PRODUCER makes a call to MSG_HANDLER.SEND, the accept body executes. MSG_HANDLER receives the character CH and stores it in the local variable MSG. After the accept statement concludes MSG_HANDLER and PRODUCER continue execution.

Synchronization Constructs

Three different forms of the select statement are available to control task synchronization. The first is the *selective-wait*, which is used to coordinate among possibly several accepts for a task. The selective-wait provides the ability to conditionally accept only when there is a pending entry call or to wait for an entry call for a prespecified amount of time. *Timed and conditional entry calls* are the remaining two forms of the select statement. These forms allow for no waiting for a prespecified wait when issuing an entry call.

The Selective-Wait

The selective-wait statement is made up of one or more alternatives each specifying actions. One alternative is selected and executed each time the construct is encountered. Alternatives can be included for conditionally/immediately accepting entry calls or for specifying contingency actions if an entry call cannot be accepted.

One or more accepting alternatives can exist in a select. Accept alternatives may be guarded allowing the programmer to specify the task conditions needed to execute the accept. Statements may also be placed following the accept body of a select alternative. These statements are executed after the rendezvous has completed.

Contingency alternatives describe what to do when an accept alternative cannot be executed. If they are present, contingency alternatives may take one (only) of the following forms: (1) an else, (2) one or more delay alternatives, (3) a terminate alternative. As with accept alternatives, the delay and terminate alternatives may be guarded by a conditional. The delay and else may also include statements to be executed when selected.

```
task body BUFFER is;
   POOL_SIZE : constant INTEGER := 100;
   POOL : array(1 .. POOL_SIZE) of MESSAGE;
   COUNT : INTEGER range 0 .. POOL_SIZE := 0;
   FRONT, REAR : INTEGER ranoe 1 .. POOL_SIZE := 1;
begin
  loop
    select
      when COUNT < POOL_SIZE =>
        accept SEND(C : in MESSAGE) do
          POOL(REAR) := C;
        end;
        REAR := (REAR + 1) mod POOL_SIZE;
        COUNT := COUNT + 1;
      or when COUNT > 0 =>
        accept RECEIVE(C : out MESSAGE) do
          C := POOL(FRONT);
        end;
        FRONT := (FRONT + 1) mod POOL_SIZE;
        COUNT := COUNT - 1;
      end select;
    end loop;
end BUFFER;
```

Figure 2. BUFFER using a selective wait to queue messages.

Execution of a select begins by evaluating the guards in an undetermined order. All alternatives having a true guard and alternatives having no guards are said to be open. If one or more accept alternatives are open and also have queued entry calls then one is arbitrarily selected and executed. If an immediate rendezvous is not possible then selection depends on the contingency alternative.

No (open) contingency alternative: If there are open accepts then the task waits until an entry call is made to one of the open entries. When there are no open alternatives the exception PROGRAM_ERROR is raised.

An else is present: The else part is executed if no open accept alternative can be immediately selected.

One or more delay alternatives are present: Delays specify a time to wait for an entry call and actions to be performed should the wait time expire. The open delay alternative with minimal time delay is selected (arbitrarily if more than one have the same delay) if no entry call is made to an open accept beforehand.

Terminate: An open terminate is selected and the task is terminated if the language-defined conditions for termination are satisfied before an entry call is made on an open accept.

Figure 2 shows how the BUFFER task presented previously can use the selective-wait to allow messages to be queued as they are transmitted from the PRODUCER to the CONSUMER. This example is similar to that appearing in the *Ada Language Reference Manual* [4].

Timed Entry Call Statement

This form of the select provides the programmer with the ability to specify, upon making an entry call, the time to wait for a rendezvous to begin. The syntax of the timed entry call is

```
TIMED_ENTRY_CALL ::=
  select
    ENTRY_CALL_STATEMENT
    [SEQUENCE_OF_STATEMENTS]
  or
    DELAY_STATEMENT
    [SEQUENCE_OF_STATEMENTS]
  end select;
```

If possible, the rendezvous is initiated before the time specified in the delay statement. If this is not possible, the statements following the delay are executed and the entry call is canceled. Ada also provides a continental entry call as another form of the select statement. The conditional entry call has the same semantics as the timed entry call except that no time delay is specified. If a rendezvous cannot immediately begin then the else part is executed.

THE INTEL 80286

The iAPX286 from INTEL is an 8086 upward compatible VLSI microprocessor system based on the 80286 CPU. The iAPX286 provides many hardware features for today's large multiuser and real-time multitasking systems. Hardware protection, virtual address spaces of up to 1 gigabyte per task, mutual exclusion of these address spaces, on-chip memory management, and virtual memory support are a few of the features of this system.

The 80286 CPU consists of four separate processing units. An address unit performs pipelined calculation of effective addresses while making hardware protection checks based on cache protection information. A bus unit manages a demultiplexed bus structure using pipelining techniques to provide a bus bandwidth of 10 megabytes per second using a 10-MHz clock [3]. An instruction unit decodes instruction codes received from the bus unit and manages a prefetch queue. The execution unit executes decoded instructions that are received from the instruction unit.

```
                      bytes
+----------------+
| INTEL reserved |    7-6
+--------+-------+
| access | base  |    5-4
+--------+       |
|    address     |    3-2
+----------------+
|      size      |    1-0
+----------------+
```

Figure 3. Segment descriptor for the INTEL 80286.

Real and Protected Address Modes

The iAPX286 can run in either of two addressing modes. In the real address mode the virtual address space is identical to the physical address space, and programs manipulate physical memory locations. Up to 1 megabyte of physical memory may be addressed, and in this mode the 80286 appears to be a fast 8086.

Protected Virtual Address Mode

In protected virtual address mode, the 80286 provides additional features while maintaining close to the same baseline architecture as the 8086. The major additional features provided are virtual address translation, memory protection, and multitasking support.

The total virtual address space for a task is 1 gigabyte, which is mapped into 16 megabytes of physical memory. Each task has its own Local Descriptor Table (LDT). Additionally, there is a Global Descriptor Table (GDT) common to all tasks. Information may be referenced only through these descriptor tables, thus providing mutual exclusion of local address space.

Descriptor Tables

A descriptor table may contain two types of descriptors, segment descriptors and control descriptors. A segment descriptor is shown in Figure 3. Bytes 6 and 7 of the descriptor are INTEL reserved for upward compatibility, and byte 5 is the access rights byte. The 24-bit base address field (bytes 2–4) points to the start of the segment in the 16-megabyte physical memory. The 16-bit size field indicates the current length of the segment and is used to insure that all references are in range.

The second type of descriptor in a descriptor table is a control descriptor. Control descriptors reference special system data segments. These segments include descriptor tables, descriptors for task state segments (segments that define the current state of a task), and call the task gates which are used for transfer of control. Control descriptors are differentiated from segment descriptors by the access rights byte. Although control descriptors may reside in the GDT, they are not accessible to tasks.

Address Translation

A task references the location of a byte of memory using a 32-bit virtual address, although the instruction itself may not contain all 32 bits. Of the 32 bits, 16 are used to select a segment descriptor and 16 are used to index into the selected segment. The top 16 bits are the value of the segment register and select a descriptor from either the GDT or the LDT of the task. A 16-bit offset into the segment is added to the base address field of the descriptor to complete the 32-bit virtual to 24-bit physical address translation.

The Register Set

The register set of the 80286 consists of general-purpose registers, segment registers, status registers, and special-purpose registers. There are eight (16-bit) general-purpose registers, four of which are addressable as byte or word registers, and the other four of which are default registers for operands of many instructions and are addressable only as word registers. Tasks running in the 80286 environment usually consist of several code segments, data segments, and a stack segment. The four segment registers (CS, DS, ES, and SS) are used to provide immediate access to four of the segments of a task. Each is a pointer into the descriptor tables for the currently executing task. Most memory references are simple 16-bit offsets to the segments referenced by the DS, ES, and SS registers. If the data desired do not reside in a currently referenced segment, a full 32-bit address is needed to first reload the segment register, and then offset into the newly referenced segment. The SS can be loaded explicitly for dynamic stack reconfiguration. There are three (16-bit) status and control registers including an instruction pointer, a machine status word, and a flags register. Finally, there are three (16-bit) special-purpose registers which keep track of the locations of the current local descriptor table (LDTR), the global descriptor table (GDTR), and the segment that contains information on the state of the current task (TR).

Task Management

Each process running in an 80286 environment is called a task, and among other features, the

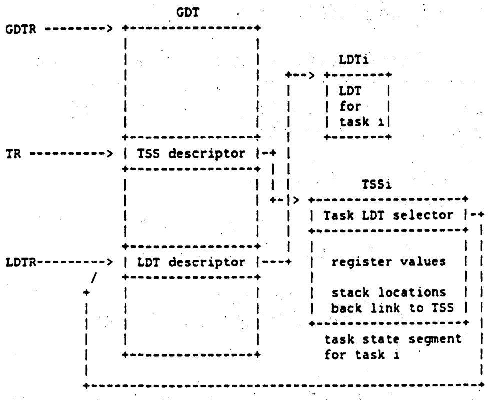


Figure 4. Task management information.

system provides an efficient task switch mechanism. Each task is defined by a special segment known as a Task State Segment (TSS) as shown in Figure 4. Descriptors for the TSSs uniquely identify a task and are stored in the GDT. A special-purpose register, the Task Register (TR), always contains a selector into the GDT for the current TSS descriptor. The TSS defines the execution state of a task through register values, stack locations, and a backpointer to the previous task. The address space of a task is also defined in its TSS. The TSS contains a selector into the GDT where a descriptor for the task's LDT can be found.

The Target Architecture

The target system consists of several 80286 processors, denoted (Pn), each with its own local memory (Ml) as shown in Figure 5. All processors address a global memory (Mg) through a high-speed

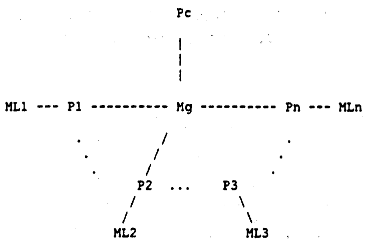


Figure 5. Hypothetical multiprocessor with shared memory.

hardware connection. The address space (both physical and virtual) of each processor has local memory and global memory addressable through its LDT and GDT, respectively. Additionally, there is a controlling processor (Pc) dedicated to running system scheduling and other systemwide functions. This processor has a link to all other 80286 processors that allows it to force a task switch (i.e., interrupt and load the CS:IP register pair). As in all tightly coupled systems, memory contention can be a major bottleneck. Without precluding other methods, global memory synchronization is assumed through the use of the LOCK prefix for memory accesses (assuming a multiported RAM for global memory). The LOCK prefix asserts a bus lock for the duration of the memory access. Using this approach, global memory contains (1) the system Global Descriptor Table (GDT), (2) all local descriptor Tables (LDTs), (3) all Task State Segments (TSSs), (4) the system scheduler, (5) code segments for each accept statement, (6) segments for all identifiers referenced by any accept statement, and (7) the data structures used for controlling synchronization.

EXECUTING SYNCHRONIZATION CONSTRUCTS

In this section we present the code needed to implement synchronization among tasks of an Ada main program. The technique provides for an Ada program to be distributed across the target architecture such that tasks of the program may execute on different processors. A single task, however, is not distributed across processors, and multiprocessing is not precluded as a level of control above that described here.

Data Structures to Support Synchronization

The data structures for managing a task's entries are accessed by both the caller and the called tasks in a controlled manner to provide for synchronization. A handshaking mechanism is used to initiate the rendezvous, whereby the task arriving at the synchronization point last will execute the accept statement code. The data structures coordinating the rendezvous are kept in global memory where they may be accessed by tasks as necessary through their GDT. A segment of code, called ENTRY_CALL, is executed by the calling task to determine whether it must be queued for the call or the accept body may be immediately executed. In a manner similar to

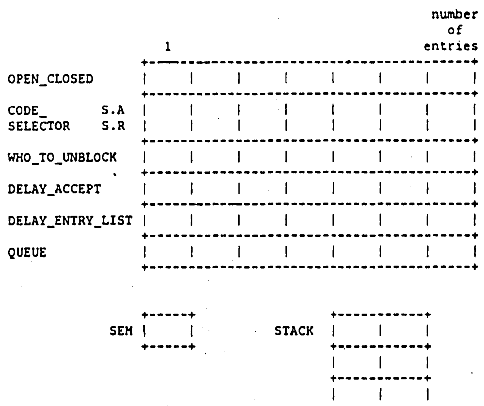


Figure 6. Rendezvous management data for a task.

Habermann's solution, the accept code is treated as a callable procedure, but in our solution either the calling or the called task may execute the accept statement. To make the procedure visible to either task, the procedure is placed in global memory together with all identifiers it references. In the accepting task, a section of code named SELECTIVE_WAIT is executed to perform the corresponding actions for the called task. SELECTIVE_WAIT is executed each time an accept/select is encountered. While all accept statement code must be placed in global memory, any select code that is not textually contained within an accept statement may be placed in separate local code segments.

There are three compiler-generated data structures for each task that are used to provide the synchronization necessary to implement the rendezvous mechanism. The structures, as shown in Figure 6, consist of a table of entry variables, a synchronization semaphore, and a stack of tasks and return pointers.

The task's table contains state information about each entry in the task. The table's OPEN_CLOSED field is used to indicate whether an accept statement's guard is true. Since a task may have many accept statements for a single entry, the field CODE_SELECTOR is used to indicate which accept is currently active. Select statements having delay alternatives use the Boolean variable DELAY_ACCEPT to indicate a call must arrive within a specified time. DELAY_ENTRY_LIST is a list of tasks having a conditional or time entry call outstanding for the entry. Since accepts may be nested and since either the calling or the called task may execute the

231543-6

accept statement code, the variable WHO_TO_UNBLOCK is needed to indicate the task to be activated after an accept statement completes. An entry queue, QUEUE, is kept for each entry in a task to retain entry calls that occur prior to a matching accept.

The task's semaphore, called RENDEZVOUS.SEM, is used to control access to the synchronizatiion data structure for the task. A stack (RENDEZVOUS.STACK), whose depth is equal to the maximum nesting level for accept statements, is used to coordinate returns from nested accepts. Initially, all OPEN_CLOSED are CLOSED, all Booleans are FALSE, the queues are empty, all code selectors are null, the stack is empty, and the rendezvous semaphore is AVAILABLE.

The data structures are grouped together in a single record in global memory. Each task has its own record, which is accessed through a selector called RENDEZVOUS. A task's RENDEZVOUS selector is made available to all other tasks that may call it through their GDT.

The algorithms that manipulate this data structure are expressed in Ada-like code. Queue, list, and stack operations are assumed to exist as needed. Exclusive use of the RENDEZVOUS data structure is enforced using *P* and *V* operations, which we assume are implemented without scheduler interactions (i.e., busy wait for *P*). BLOCK and UNBLOCK, however, assume system scheduler intervention.

Accept and Selective-Wait Statements

An accept statement may be either part of a selective-wait statement or a free-standing accept statement. To avoid considering the two cases separately, the free-standing accept is treated as being in a select statement with only one alternative. This simplifies the design and removes the need for additional synchronization overhead.

The code associated with each accept statement in a task body is placed in a global code segment called S.A(i), where S, A, and i indicate the accept statement part of the *i*th alternative within the selective-wait S. All identifiers referenced by an accept statement S.A(i) are also placed in a global segment. Access to the accept statement code and the needed identifiers is provided to the accepting task through run-time initialization of its GDT. When a calling task needs access to the synchronization code, it is provided through the RENDEZVOUS record's CODE_SELECTOR field for the task.

If an accept alternative within a selective-wait contains code that is not in the accept statement (i.e., the rest of a select alternative), that code is placed in a local code segment called S.R(i). S.R(i) is the rest of the *i*th alternative within the selective-wait S. S.R(i) is always executed by the accepting task and is also stored in the CODE_SELECTOR field.

The prologue code, called SELECTIVE_WAIT, is executed by a task each time it encounters an accepting construct. The code for SELECTIVE_WAIT is shown in Figures 7 and 8. The routine is passed a selector for the compiler-generated synchronization data structures (RENDEZVOUS) and information about the select statement consisting of (a) for each accept alternative its accept statement [S.A(i)]; (b) for each accept alternative the rest of the alternative [S.R(i)] (Note: if an alternative has no "rest" part S.R(i) is the first statement following the selective wait); (c) code selectors for the body of each delay or else contingency alternative [S.D(i), S.E]; (d) the code for evaluating the Boolean expressions guarding accept, delay, and terminate alternatives.

SELECTIVE_WAIT first performs a *P* operation on RENDEZVOUS.SEM to prohibit other tasks from accessing the synchronization data for the task. SELECTIVE_WAIT then evaluates the guard for each alternative. A missing guard is taken as true.

```
procedure SELECTIVE_WAIT(RENDEZVOUS,S.A,S.R,S.D,S.E,GUARDS) is

  begin
    P(RENDEZVOUS.SEM)     --don't let anyone else access the
                          --synchronization data until its
                          --update is complete.
    CALL EVALUATE(GUARDS)
    for ALL i HAVING TRUE GUARDS loop
         RENDEZVOUS.OPEN_CLOSED(i):=OPEN;
    end loop;
    case SELECT ENTRY SUCH THAT:
        (NOTEMPTY(QUEUE(ENTRY)) AND OPEN_CLOSED(ENTRY)=OPEN) is
                            --if multiple entries satisfy the
                            --condition then arbitrarily select
                            --one.
     when ENTRY FOUND =>    --entry call before accept
         for ALL ENTRIES i loop
           RENDEZVOUS.OPEN_CLOSED(i):=CLOSED;
         end loop;
         ACCEPT_STATEMENT := RENDEZVOUS.S.A(ENTRY);
         DEQUEUE(CALLER_TSS,ARGUMENTS,CALLER_RETURN,
                FROM=> RENDEZVOUS.QUEUE(ENTRY));
         PUSH(CALLER_TSS,CALLER_RET,ON=>RENDEZVOUS.STACK);
         if NOT ISEMPTY(RENDEZVOUS.DELAY_ENTRY_LIST(ENTRY))
           then CANCEL_TIMER(CALLER_TSS);
                REMOVE(CALLER_TSS,FROM=>
                         RENDEZVOUS.DELAY_ENTRY_LIST(ENTRY));
         end if;
         V(RENDEZVOUS.SEM);
         CALL ACCEPT_STATEMENT(ARGUMENTS);
         P(RENDEZVOUS.SEM)
         (CALLER_TSS,CALLER_RETURN) := POP(RENDEZVOUS_STACK);
         V(RENDEZVOUS.SEM)
         UNBLOCK(CALLER.TSS,AT=>CALLER.RET)
```

Figure 7. Accept prologue for entry call before accept.

231543–7

```
   when ENTRY NOT FOUND AND NO OPEN CONTINGENCY ALTERNATIVES =>
      for ALL i SUCH THAT: OPEN_CLOSED(i)=OPEN loop
         RENDEZVOUS.CODE_SELECTOR(i):=(S.A(i),S.R(i));
         RENDEZVOUS.WHO_TO_UNBLOCK(i) := ME.TSS;
      end loop;
      V(RENDEZVOUS.SEM);
      BLOCK;

   when ENTRY NOT FOUND AND OPEN CONTINGENCY ALTERNATIVES =>
      case ON TYPE OF OPEN CONTINGENCY ALTERNATIVES is
         when ELSE | TERMINATE =>
            for ALL ENTRIES i IN RENDEZVOUS DATA loop
               RENDEZVOUS.OPEN_CLOSED(i):=CLOSED;
            end loop;
            V(RENDEZVOUS.SEM);
            if TYPE = ELSE then goto S.E;
                           else terminate(me);
            end if;

         when OPEN DELAY ALTERNATIVES =>
            for ALL ACCEPT GUARDS i THAT WERE TRUE loop
               RENDEZVOUS.DELAY_ACCEPT(i) := TRUE;
               RENDEZVOUS.CODE_SELECTOR(i) := (S.A(i),S.R(i));
               RENDEZVOUS.WHO_TO_UNBLOCK(i) := ME.TSS;
            end loop;
            for ALL OPEN DELAY ALTERNATIVES i loop
               EVALUATE_SELECT(DURATION(i),WHICH,TIME);
                          --evaluate durations(arbitrarily)
                          --selecting the minimum
            end loop;
            START_TIMER(TIME,WHICH,WAKEUP,RENDEZVOUS.SEM)
            V(RENDEZVOUS.SEM);
            BLOCK;
WAKEUP(WHICH):               --P(SEM) obtained with wake-up.
            for ALL ENTRIES i IN RENDEZVOUS DATA loop
               RENDEZVOUS.DELAY_ACCEPT(i) := null;
               RENDEZVOUS.OPEN_CLOSED(i) := CLOSED;
               RENDEZVOUS.WHO_TO_UNBLOCK(i) := null;
               RENDEZVOUS.CODE_SELECTOR(i) := (null,null);
            end loop;
            V(RENDEZVOUS.SEM);
            RETURN to S.D(which)
         end case;        --on contingency alternative
      end case;           --on ENTRY found
end SELECTIVE_WAIT;
```

Figure 8. Accept prologue when a rendezvous is not immediate.

For each accepting alternative present in the construct, its OPEN/CLOSE indicator is bound.

Next, an examination is made to determine whether any open accept alternatives have calling tasks queued on their wait list. If a calling task is found then the code of Figure 7 is executed, which causes the task calling SELECTIVE_WAIT to execute the accept statement code. If a calling task is not found then the code of Figure 8 is executed. In this case the task acting on behalf of the caller will execute the accept statement code.

If a waiting caller is found (see Fig. 7) then the necessary initializations are made, the exclusion on the rendezvous data is released, and the accept statement code is invoked as a subroutine of SELECTIVE_WAIT. After the rendezvous code is completed, exclusion is again obtained on the synchronization data structures to determine the identity of the calling task. This task is then unblocked and the exclusion is released.

If no open accept alternatives have queued entry calls (see Fig. 8) and no contingency alternatives exist (else, delay, or terminate), the accepting task must wait for an entry call. In this case the accepting task will not execute the rendezvous code, so the CODE_SELECTOR fields of the rendezvous data structure are set to give a future calling task access to the code. The accepting task then blocks itself until a caller arrives. The caller will execute the accept statement and perform the unblock (indicating where control is to continue by specifying AFTER_ACCEPT).

If no calling tasks were found for an open accept alternative and open contingency alternatives are present then Ada select semantics depend on the type of contingency alternative. When an *else* or *terminate* alternative exists, the synchronization data structures are reset to indicate that the selecting task is no longer able to rendezvous. Setting all OPEN_CLOSED fields to CLOSED indicates that the task is not accepting. At this point the data structures are released and the action associated with the contingency is performed. That is, the *else* clause is executed or the conditions for *termination* are examined.

When one or more delay alternatives are open then the task must block for the shortest duration specified. Delays are accomplished by calling the routine

START_TIMER(HOW_LONG,WHICH,
WHERE_TO_WAKE,SEM)

which will unblock the task at WAKEUP after TIME has expired. The argument WHICH is provided so the task may determine the delay alternative that has expired. If the timer expires, control is returned indicating the timer has completed. If an open entry is called during the delay, the calling task will cancel the delay, execute the rendezvous code, and unblock the called task at the appropriate S.R(i).

As an indivisible part of waking a task up on completion of the timer, exclusion is obtained on the semaphore SEM. Treating waking up a task and obtaining exclusion on the rendezvous data as a single operation is one example of indivisible operations that must be performed. In this case the operation is necessary to prohibit another rendezvous statement from altering the data structure between the wake-up and obtaining the semaphore. Another example occurs when a *V* operation is followed by a BLOCK.

Entry Calls

According to our design, the last task to arrive at the rendezvous, whether it is the calling or accepting task, is responsible for executing the synchronized code. Since the accept statement itself is textually a part of the called task, certain provisions must be made to allow it to be executed by the caller. Aside from placing the accept statement code and the identifiers it references in global memory, any interrupts that occur during the rendezvous must be held. Additionally, Ada task priorities (defined by a PRAGMA) and the rules for prioritization are not considered by our algorithms. Ada also specifies that any exceptions raised during a rendezvous should be raised in both tasks. The run-time support for exceptions can most conveniently accomplish this through access to the rendezvous management data.

Ada provides conditional and timed entry calls, in addition to the standard entry call. Semantically, a conditional entry call can be expressed as a timed entry call with a zero delay. A standard entry call could also be represented as a timed entry call with an infinite delay, and this simplification is made so that a single algorithm can be presented.

```
procedure ENTRY_CALL (TASK,ENTRY,ARGUMENTS,
                      AFTER_CALL,DELAY_STATEMENT) is

begin
   RENDEZVOUS := selector to access rendezvous data of the
                 TASK owning the entry;
   P(RENDEZVOUS.SEM);
   if ISEMPTY(RENDEZVOUS.QUEUE(ENTRY)
            and RENDEZVOUS.OPEN_CLOSED(ENTRY)=OPEN)
      then    --Accept before call.
         ACCEPT_STATEMENT := RENDEZVOUS.S.A(ENTRY);
         PUSH(RENDEZVOUS.WHO_TO_UNBLOCK,RENDEZVOUS.S.R(ENTRY),
              ON=>RENDEZVOUS.STACK);
         for ALL ENTRIES i IN RENDEZVOUS DATA loop
            RENDEZVOUS.OPEN_CLOSED(i) := CLOSED;
            RENDEZVOUS.CODE_SELECTOR(i) := (null,null);
            RENDEZVOUS.WHO_TO_UNBLOCK(i) := null;
            if RENDEZVOUS.DELAY_ACCEPT(i)
              then CANCEL_TIMER(RENDEZVOUS.WHO_TO_UNBLOCK,i);
                   RENDEZVOUS.DELAY_ACCEPT(i) := FALSE;
            end if;
         end loop;
         V(RENDEZVOUS.SEM);
         CALL ACCEPT_STATEMENT(ARGUMENTS);
         P(RENDEZVOUS.SEM);
         (ACCEPTOR,ACCEPTS_REST) := POP(RENDEZVOUS.STACK);
         V(RENDEZVOUS.SEM);
         UNBLOCK(ACCEPTOR, AT=>ACCEPTS_REST);
         GOTO AFTER_CALL;
      else
         ENQUEUE(ME.TSS,ARGUMENTS,AFTER_CALL,
                 ON=>RENDEZVOUS.QUEUE(ENTRY));
         ADDLIST(ME.TSS,RENDEZVOUS.DELAY_ENTRY_LIST(ENTRY));
         START_TIMER(DURATION,ENTRY,WAKE_UP,RENDEZVOUS.SEM);
         V(RENDEZVOUS.SEM);
         BLOCK;
WAKE_UP(ENTRY):                --if awakened here then wait time
                               --expired and exclusive access to
                               --data has been obtained.
         REMOVE(ME.TSS,FROM=>RENDEZVOUS.DELAY_ENTRY_LIST(ENTRY));
         DEQUEUE(ME.TSS, FROM=>RENDEZVOUS.QUEUE(ENTRY));
         V(RENDEZVOUS.SEM);
         GOTO DELAY_STATEMENT or AFTER_CALL;
   end if;
end ENTRY_CALL;
```

Figure 9. Prologue code for entry calls.

The prologue code for all three forms of entry call is shown in Figure 9. This procedure, called ENTRY_CALL, has parameters indicating the called task, the entry within the task, arguments to the call, the first statement after the call, the time duration to wait for an accept, and, if present, the delay statement. Access to the global segment containing the called task's rendezvous management data is established using TASK. This selector is called RENDEZVOUS. Exclusive access to the data is next obtained. To determine whether the caller or the accepting task will execute the rendezvous, the proper entry QUEUE and OPEN_CLOSED indicator of the called task are examined. The caller will execute the rendezvous if the QUEUE is empty and the indicator is OPEN. Otherwise the caller places itself on the queue, releases exclusion on the rendezvous data, and blocks itself. Before calling BLOCK, a timer is set to indicate how long a matching accept is to be awaited.

If the caller will execute the accept statement then the rendezvous data is set accordingly. Before releasing exclusion on the rendezvous data, the caller checks whether a delay contingency is active in the called task. If one is active all timers in the called task are disabled. After completing the accept statement, the rendezvous data are used to determine the appropriate return point within the called task, and the task is unblocked.

Parameters and Shared Variables

Parameters

The Ada language allows for various parameter implementation mechanisms to be used for subroutine linkage. Parameters used in tasking follow the same rules. Parameter modes of IN, OUT, and IN OUT may all be used to pass information between the calling and the called tasks. For IN OUT parameters, either *copy restore* or *call by reference* may be used. Any Ada program that would execute differently for one implementation than the other is erroneous. Although parameters are not included explicitly in the algorithms for SELECTIVE_WAIT and ENTRY_CALL, they may best be handled using call by reference.

Parameters of mode IN use only a copy. Within ENTRY_CALL, IN only arguments would be evaluated and copied to either the appropriate wait queue or the parameter. Arguments to a rendezvous may be kept in local memory if they are copied by the synchronization primitives. If an entry call exists within an accept statement, however, recall that all identifiers referenced from within an accept must be kept in global memory.

To implement parameters of mode OUT or IN OUT using call by reference requires that all arguments reside in global memory. ENTRY_CALL must convey a selector to the global segment containing the arguments. If the call needs to be queued then the selector is placed on the queue, otherwise the selector is conveyed directly to the accept statement.

Shared Variables

Using the multiprocessor 80286 architecture, the synchronization primitives suggested, and the parameter mechanisms discussed above, an excessive amount of data copying is required to transfer a message from the calling task to the caller. For example, in the buffered message task of Figure 2, the message to be transferred is an IN parameter to SEND and an OUT parameter from RECEIVE. Assuming the select is encountered before the entry call to SEND and RECEIVE, the following copies of the message would be made in its transmission: (1) from sender's local memory to the parameters in global memory, (2) from the parameter of accept SEND to the buffer task's POOL, (3) from POOL to the global segment containing the consumer task's argument. These copies represent the best case, as an additional copy is required when the entry call to SEND must be queued. The additional copy is placed on the wait queue for SEND.

While message passing using an IN and OUT parameter has the advantage that the message can be created in local storage, the cost of copying large messages can be reduced. If SEND's parameter mode were IN OUT, global storage would be used to create the message, but the first copy could be eliminated.

A more desirable solution can be found if we assume that only one producer and one consumer use a single buffer, and by generalizing the Ada definition of synchronization points between tasks. In this instance a solution in which messages are both created and consumed using the same global memory can be constructed. This solution, which does not require any copying of the message for single-processor implementations and distributed implementations with shared memory, is shown in Figure 10. No copies of the message are required to transfer it from the producer to the consumer. SEND and RECEIVE are used to synchronize access to messages and to allow messages to be queued/buffer.

In Figure 10 the following data sharing takes place: (i) The producer task shares REAR with the buffer task. Notice that both tasks may examine REAR between synchronization points, but REAR is only updated by the buffer task during synchronization. (ii) The consumer task shares FRONT with the buffer task. Again, notice that both the consumer and the buffer may examine FRONT between synchronization points, but FRONT is only updated by buffer during synchronization with the consumer.

```
--shared objects
     POOL_SIZE : constant INTEGER := 100;
     POOL  : array (0 .. POOL_SIZE-1) of MESSAGE;
     FRONT, REAR ¬ INTEGER range 0 .. POOL_SIZE-1 := (0,1);

     producer task:
          loop
              --load POOL(REAR) with the message;
              BUFFER.SEND;
              exit when no more messages;
          end loop;

     consumer task:
               BUFFER.RECEIVE;
               --use message in POOL(FRONT);
               exit when no more messages;
          end loop;

     buffer task:
          loop
            select
               when ((REAR+1) mod POOL_SIZE) /= FRONT =>
                  accept SEND
                     REAR := (REAR+1) mod POOL_SIZE;
                  end;
               when (FRONT+1) mod POOL_SIZE) /= REAR =>
                  accept RECEIVE
                     FRONT := (FRONT+1) mod POOL_SIZE;
                  end;
            end select;
          end loop;
```

Figure 10. Message passing without copying the message.

The producer and the consumer tasks share access to the queue of messages in POOL. A single element of POOL may be updated by the producing task and read by the consuming task between synchronization points of the tasks (where *synchronization points* are defined by the Ada language [4]). Although Ada does not allow this type of sharing, these two tasks (and their access to POOL) are properly synchronized indirectly by the buffer task. A distributed implementation without shared memory could maintain integrity of the shared variable POOL by examining access at every synchronization point. By recording reads and updates to shared data, and by comparing these records at synchronization points, proper use of this generalized form of shared data could be enforced.

If POOL, FRONT, and REAR are placed in global memory, each task/processor may access messages efficiently. Although their placement in global memory places an additional burden on its use, a majority of message-passing applications require significant local processing in creating and consuming messages. One such example, although not an embedded application, is a compiler's lexical and syntactic analysis phases. In this case, messages are tokens or streams of tokens that are recognized and created by the lexical analyzer. The syntax analyzer receives the tokens and uses them to produce input to the next compilation phase. In this case, the tokens represent the input and output to the phases with significant additional processing required to create and use the tokens.

SUMMARY

An important criticism of Ada tasking is that transmission of data from a sender task to a receiver task requires excessive scheduler interactions. Habermann [2] has offered an implementation technique that requires fewer scheduler interactions than required in a straightforward implementation. Habermann reduces interactions by making the accept statement code a subroutine invoked by the calling task. This approach correctly assumes that a majority of the applications using tasking will be arranged in a manner similar to the BUFFER task of Figure 2. The property relied upon is that the accepting task will arrive at the rendezvous before any entry calls have been queued. In this article we present a technique to execute the accept statement code. This approach minimizes scheduling requirements independent of whether the calling or accepting task arrive at the rendezvous first.

Of critical importance to the use of Ada in embedded applications is the implementability of tasking in mutliprocessor architectures. While the techniques this article presents do not apply to loosely coupled multiprocessor systems, the example architecture we discuss shows how tasking can be implemented efficiently for multiprocessors with shared memory. For systems in which heavy message traffic occurs between tasks, such as real-time embedded applications, minimizing the overhead in message transmittal is equally as important as minimizing scheduler interactions. This article presents a form of message passing that would minimize message overhead. The solution requires sharing memory among sending and receiving tasks in a manner that violates Ada assumptions about shared variables, but which is implementable with reasonable efficiency.

References

1. D. Cornhill, "A survivable distributed computing system for embedded application programs written in Ada," *Ada Letters*, **III** (3), 79–87 (December 1983).
2. A.N. Habermann, "Efficient implementation of Ada tasks," Technical Report CMU-CS-80-103, Department of Computer Science, Carnegie-Mellon University, 1980.
3. INTEL Corp., *Introduction to the iAPX286*, Santa Clara, CA, 1982.
4. *Reference Manual for the Ada programming language*, ANSI/MIL-STD 1815 A, U.S. Department of Defense, January 1983.
5. E.S. Roberts, A. Evans, Jr., and C.R. Morgan, "Task management in Ada—a critical evaluation for real-time multiprocessors," *Software—Practice and Experience*, **11**, 1019–1051 (1981).

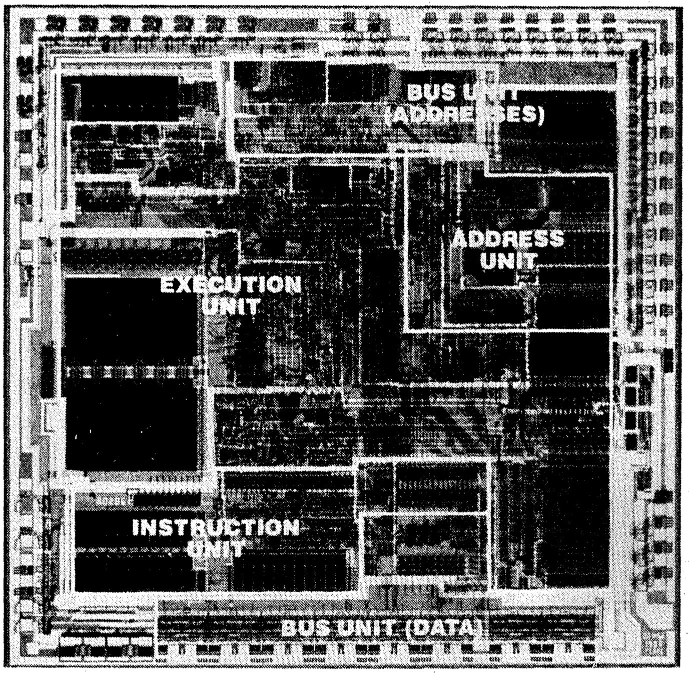
Ada Capabilities for Today's Microprocessors

An Ada implementation demonstrates how the DOD's high-order language can take full advantage of advanced microprocessors in real-time applications.

By Liz Parrish

The primary goal pursued by the Department of Defense in developing the Ada high-order language (HOL) is to reduce software life cycle costs. The DOD is attempting to do this by using one HOL that ensures standardization in all areas—design, documentation, interfaces to the outside world, training, testing, and capabilities and functionality available to programmers. Ada's portability and maintainability, as well as the ongoing emphasis on validation to ensure strict compliance with the language definition, also contribute to reducing the software costs that play such a major role in today's embedded systems programs. In particular, development costs can be reduced by using proven programming practices such as top-down design, modular programming, information hiding and machine independence.

Despite these benefits, many people are still asking, "Is Ada really useful today?" But perhaps the more important question—one that more programmers, designers and managers are asking—is, "Is Ada really *usable* today?" As more compilers become validated, and as the use of Ada continues to grow, especially in embedded real-time systems, programmers are voicing some basic concerns about Ada. Specific questions include: "How does Ada work to produce portable and maintainable code?" "If I must use Ada, what advantages will I realize?" "Can Ada programs work on microprocessors in real-time applications?" "Will Ada and microprocessors continue to evolve to help solve design problems?" These questions must be resolved in order for Ada to achieve widespread acceptance.



***Anatomy of a microprocessor:** Intel's iAPX 286 microprocessor provides many optimizations to support high-order languages such as Ada.*

Ada Language Features

Ada is considered to be a derivative of and a superset to PASCAL, and thus has many of the same features. PASCAL's capabilities such as the availability of various data types (Boolean, Record, Double Float), flow control statements (Do While, Case), and block structuring have their counterparts in Ada. Ada also contains a rich set of constructs that foster the use of modern programming practices. Ada has inherent features to handle data/code encapsulation, tasking, bounds and type checking, and hardware dependencies. These features can provide an answer to the question: "What advantages will I realize by using Ada?"

Liz Parrish is product manager, Development Systems Operation, at Intel Corp., Santa Clara, Ca.

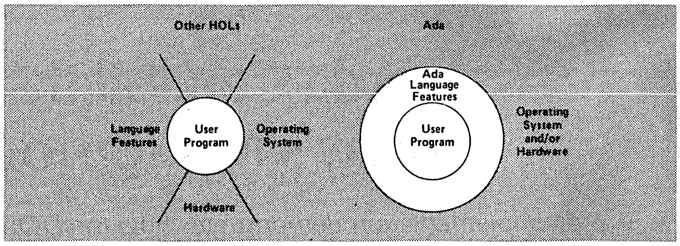
ORDER NO. 231601-001

Encapsulation, the explicit separation of items, applies to an Ada program at several levels. On the most general level, the language itself shields the application program from the underlying operating system and/or hardware. Other languages assume that certain services are provided by the operating system, and the application must access the hardware. Ada defines and provides these accesses and services, freeing the application programmer from worrying about the underlying configuration or changes to it. This independence yields true portability and maintainability, because the program's only external dependency is on the compiler itself.

Encapsulation and the information control it provides are important tools for modular, top-down design and complex system development. Encapsulation within the application provides controlled sharing and hiding of information as necessary among the various program components. This feature is implemented for code via the packages construct, and for data by strong data typing.

Packages are program elements that have two parts, a specification and a body. The body is the actual implementation code, while the specification is the external view of the module—input, output, module relationships and resources—that any other module needs to know about it. The module is really a "black box," with its specification being the description of how to interact with it. Thus, sensitive information can be controlled, and mistakes that result from misusing a module are avoided. Strong data typing and the use of enumerated data types provide built-in design and programming checks and independence from the underlying implementation.

The tasking mechanism in Ada is a primary example of an operating system service being incorporated into the language definition to ensure portability and compatibility. A multitasking capability, in which tasks are viewed as concurrent sequential processes, is inherent to real-time systems. The Ada language has a full tasking mechanism, which standardizes tasking and its uses for all Ada developers. This embedded tasking mechanism also ensures that tasking is available to allow portability of Ada programs and



Through encapsulation, *or the explicit separation of items, the Ada language shields the application program from the underlying operating system and/or hardware.*

Procedure Implementation Instructions	
ENTER	Create Procedure
LEAVE	Leave Procedure
PUSHA	Save All Registers on Stack
POPA	Restore All Registers from Stack

Protection Parameter Verification Instructions	
BOUND	Detect Value Out of Range
LAR	Load Access Rights
LSL	Load Segment Limit
VERR	Verify Read Access
VERW	Verify Write Access

String Instructions	
MOVS	Move String
CMPS	Compare String
SCAS	Scan String
LODS	Load String
STOS	Store String
REP	Repeat String Instructions to Process a Block of Memory

High-order language support instructions *offered by the 286 are particularly useful for efficient Ada implementation.*

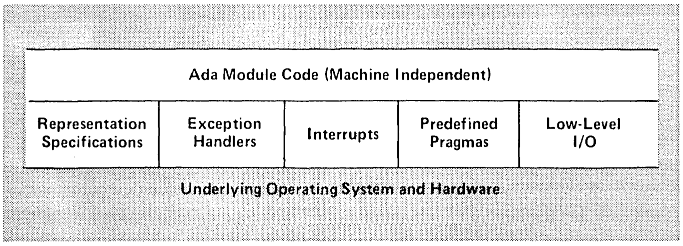
complete compatibility among programs using the same tasking implementation.

Ada has special facilities for handling machine dependencies, including representation specifications (how to represent data types in a configuration), interrupt constructs, exception handling, and certain pragmas (predefined compiler directives). There is a hierarchy of machine independence within any Ada program; this separation of machine dependencies offers the benefits of encapsulation, as well as increased reliability and efficiency. By providing a standard, controlled machine interface instead of a myriad of special cases, Ada allows these dependencies to be highly optimized, as implemented by the compiler writer, to take advantage of the underlying configuration's features. This eliminates the low-level implementation details and gives the program designer freedom to concentrate on solving any higher-level problems.

Implementing Ada on the iAPX 286

Ada provides many features and constructs that support the development of complex, real-time systems, but can the hardware match it? In particular, can today's microprocessor architectures support efficient Ada imolementations?

The answers to this question go beyond clock speed, bus bandwidth and simple throughput. As an example, Intel's iAPX 286 offers an architecture that has been optimized for general HOL development, without Ada specifically in mind. The features provided by the 286 are all necessary for an efficient Ada implementation, and Ada is the first language with the constructs to take full advantage of these features.


Ada Module Code (Machine Independent)
Representation Specifications
Exception Handlers
Interrupts
Predefined Pragmas
Low-Level I/O
Underlying Operating System and Hardware

***The machine independence hierarchy** in Ada programs is depicted. This separation of machine dependencies offers the benefits of encapsulation, reliability and efficiency.*

While the application programmer is removed from the target machine configuration, the compiler implementer works extensively with the configuration and uses a variety of methods to generate tight, efficient code. The 286 provides many optimizations—including on-chip memory management, hardware tasking, fault trapping and specific machine instructions—to support high-order languages.

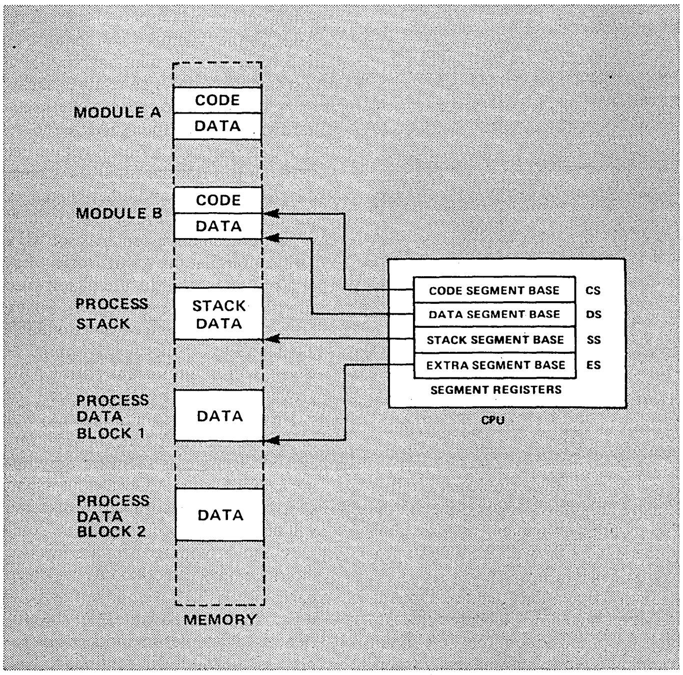
The memory management features of the 286 are useful in Ada programming. In the Protected Virtual Address mode, each task has a separate, completely independent virtual address space (up to 1 gigabyte) providing the capacity to support large Ada applications. Integrating Ada with the 286's memory management permits packages to be placed automatically in separate segments, located arbitrarily and moved in memory. This produces all of the advantages of a segmentation model designed to optimally execute code for software composed of independent modules, and software designers get the benefits of an automatic enforcement of their top-down design and program modularity.

Hardware protection features offered by the 286 apply to the full virtual address space of the system and can take several forms. On one hand, each variable-length segment contains its own program or data structure and has its own type (execute-only, execute and read, read-only, or read and write), which is checked upon each segment use; this ensures that the instruction is not in violation of the segment usage defined by the system designer. On the other hand, the task can contain code in any of four privilege levels to allow code being developed to be linked in with the rest of the system without corrupting it. Furthermore, the operating system and/or Ada runtime support is included in each task's address space, access to those services can be accomplished via a simple instruction rather than through a time-consuming context switch.

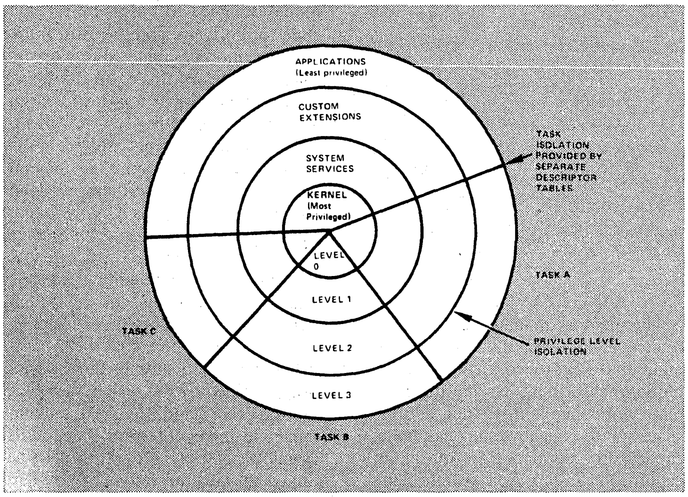
The 286's protection features provide task isolation, which is an important part of software debugging that lets the programmer isolate bugs and their effects at the applications code level. This feature is especially important to Ada developers in production systems. Hardware protection simplifies the security aspects of embedded systems by putting mission-critical or secure tasks at higher privilege levels.

The tasking model of the 286, as implemented in the hardware, directly supports the Ada tasking mechanism. The definition of tasks and the synchronization and communication mechanisms are completely compatible with Ada tasking, which offers a straightforward and efficient implementation. The task switch mechanism is also highly optimized and requires only one instruction cycle—17 microseconds using a 10-MHz 286. Thus, the hardware can support a high-performance implementation of the Ada tasking mechanism.

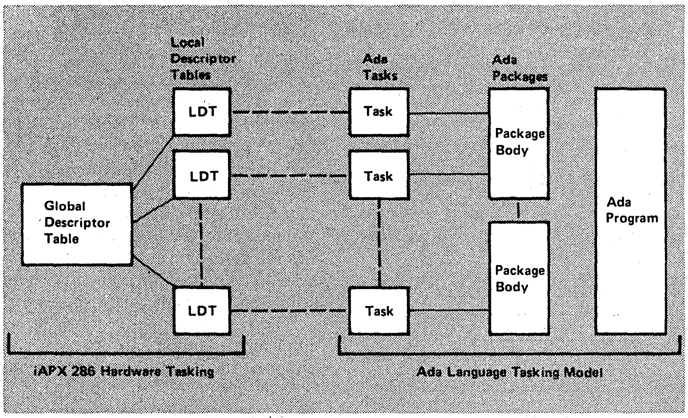
The 80287 Numeric Processor Extension provides additional processing power and functionality, and it enhances Ada code in several ways. For example, it supports the proposed



***When integrated with Ada,** the iAPX 286 memory management features permit packages to be placed automatically in separate segments, located arbitrarily and moved in memory.*


APPLICATIONS (Least privileged)
CUSTOM EXTENSIONS
SYSTEM SERVICES
KERNEL (Most Privileged)
LEVEL 0
LEVEL 1
LEVEL 2
LEVEL 3
TASK A
TASK B
TASK C
TASK ISOLATION PROVIDED BY SEPARATE DESCRIPTOR TABLES
PRIVILEGE LEVEL ISOLATION

The hardware protection mechanism *of the 286 provides task isolation—an important aspect of software debugging.*



The 286's tasking model *as implemented in the hardware directly supports the Ada tasking mechanism.*

IEEE standard 754 for binary and floating point arithmetic to ensure consistent and valid results. It supports the full range of model and safe numbers in the Ada language definition and will automatically carry out type conversions as needed (all INT types to all FLOAT types, and vice-versa). The 80287 offers any parallelism with the 286, including simultaneous execution, that is needed to take full advantage of the increased speed and higher-level interface of numeric functionality. Finally, all addressing is done via the 286 memory management, providing numerics support with the built-in protection mechanisms.

Several operations that fall into Ada low-level machine dependencies are highly optimized on the 286. Fault trapping is done automatically to enhance encapsulation and exception-handling performance. Stack and page faults are generated automatically, allowing the exception handler to correct and restart the instruction. The 286 also provides automatic bounds checking for limits, fast interrupt response and complete isolation of the exception handler.

Finally, the 286 has several machine instructions that are designed specifically to enhance HOL implementation. These instructions allow greater code density by using one of the multi-operation programming functions. Although these features are standard on the 286, Ada is the first language to offer enough built-in power to enable the generated code to take full advantage of them easily and directly.

Ada's Future

Systems development is becoming increasingly complex; the problems that will have to be solved in the years ahead are not even being considered today. Greater effort is needed to provide better and more powerful tools, to standardize and to develop methodologies that can handle these complex situations. A standard HOL such as Ada is an important part of this effort. Ada has the facilities for standardization, portability, maintainability and cost savings. Ada's rich feature set can make the most effective use of current hardware such as Intel's iAPX 286.

What lies ahead? Ada is a big first step, but it is not the "final" language; it is still maturing, and other languages will eventually be developed from it. A variety of techniques, such as software verification that proves the correctness of a program, and the use of software components (code viewed as "black boxes" with guaranteed functionality, analogous to off-the-shelf ICs in hardware), will be important parts of the growing software methodology.

However, Ada standardization will remain a central theme in real-time embedded systems software development. And just as Ada will continue to evolve, its targeted microprocessors will also evolve, becoming more complex and providing a higher level of interface and functionality to keep pace with HOL development. Ada as it is today takes advantage of the efficiencies available on modern microprocessors, and it will continue to be a tool that system and software designers can use as they strive to find the right solutions—the first time around. ■

intel®

ARTICLE
REPRINT

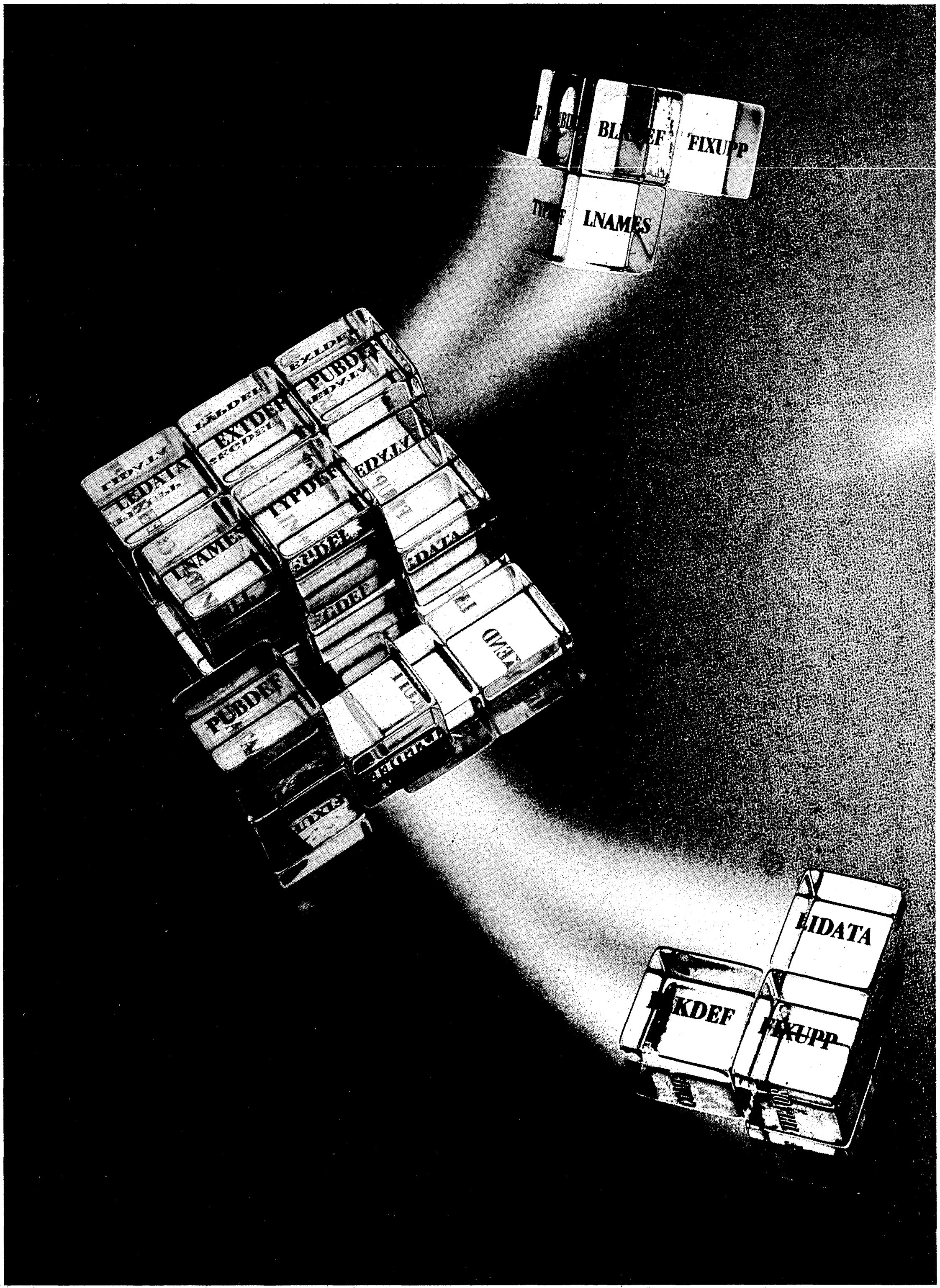
AR-501

July 1986

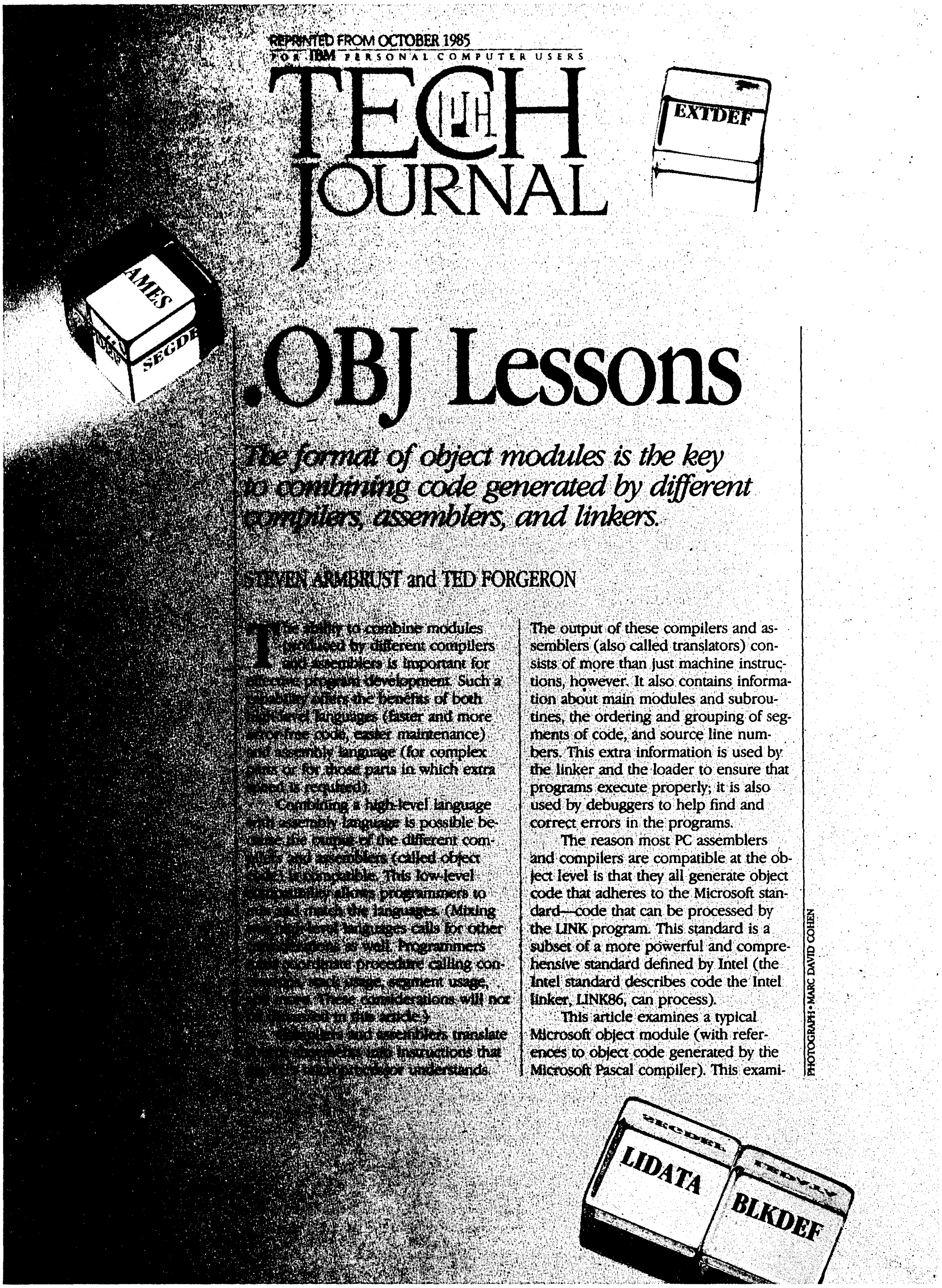
.OBJ LESSONS

STEVEN ARMBRUST AND TED FORGERON

Order Number: 280318-001


BLKDEF
FIXUPP
LNAMES
EXTDEF
PUBDEF
LEDATA
TYPDEF
LNAMES
SEGDEF
PUBDEF
MODEND
TYPDEF
FIXUPP
LIDATA
BLKDEF
FIXUPP

REPRINTED FROM OCTOBER 1985
FOR IBM PERSONAL COMPUTER USERS
PC TECH JOURNAL



.OBJ Lessons

The format of object modules is the key to combining code generated by different compilers, assemblers, and linkers.

STEVEN ARMBRUST and TED FORGERON

The ability to combine modules produced by different compilers and assemblers is important for effective program development. Such a capability offers the benefits of both high-level languages (faster and more error-free code, easier maintenance) and assembly language (for complex [illegible] or for those parts in which extra speed is required).

Combining a high-level language with assembly language is possible because the output of the different compilers and assemblers (called object code) is compatible. This low-level compatibility allows programmers to mix and match the languages. (Mixing two high-level languages calls for other considerations as well. Programmers must coordinate procedure calling conventions, stack usage, segment usage, [illegible]. These considerations will not be discussed in this article.)

Compilers and assemblers translate source statements into instructions that the [illegible] microprocessor understands. The output of these compilers and assemblers (also called translators) consists of more than just machine instructions, however. It also contains information about main modules and subroutines, the ordering and grouping of segments of code, and source line numbers. This extra information is used by the linker and the loader to ensure that programs execute properly; it is also used by debuggers to help find and correct errors in the programs.

The reason most PC assemblers and compilers are compatible at the object level is that they all generate object code that adheres to the Microsoft standard—code that can be processed by the LINK program. This standard is a subset of a more powerful and comprehensive standard defined by Intel (the Intel standard describes code the Intel linker, LINK86, can process).

This article examines a typical Microsoft object module (with references to object code generated by the Microsoft Pascal compiler). This exami-

PHOTOGRAPH • MARC DAVID COHEN

nation illustrates exactly what kind of information each object module contains and points out areas in which the Microsoft standard deviates from the way Intel originally intended for object modules to appear.

THE OBJECT MODULE

The object module is the standard unit of information produced by a program translator (such as a compiler or assembler). The module is used as input to a program linker, which combines object modules and produces an executable file (such as an .EXE file on the IBM PC). The module can also be used by a program librarian to create a library of modules for selective linking.

When a library is linked with other object modules, not all of the modules in the library are used in the creation of the final file; the linker selects those referenced by the modules being linked. Figure 1 shows this transformation of source code into object modules and finally into an executable file.

Each object module consists of groups of data called *records*. The basic structure of a record is shown in figure 2. The Record Type field is a single byte indicating a record type. The Microsoft object module format has 15 different kinds of records. Refer to table 1 for a list of those records and a brief overview of each. The complete Intel standard defines 30 types of records. Table 2 contains a list of the additional records supported by Intel.

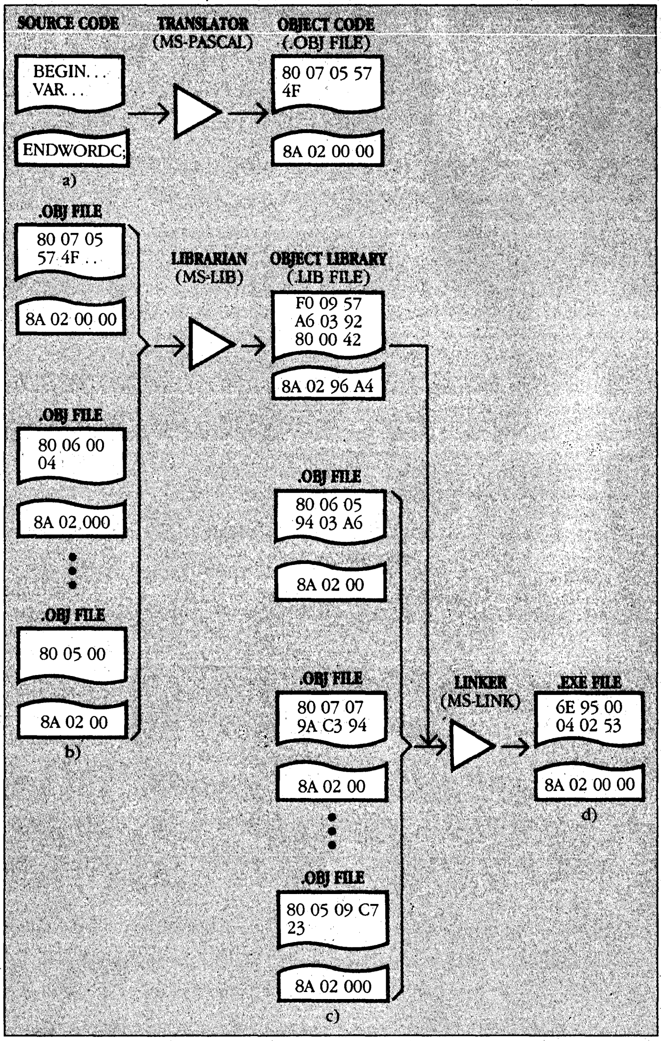
The Contents field is the meat of the record. For example, in data records, this portion can contain the actual machine code and an indication of the segment in which that code resides. The Record Length field is a two-byte value (a word) that specifies the number of bytes in the variable-length Contents field. The Checksum field helps the linker or loader identify data errors when reading the module from disk.

Every record in the object module has this basic format. The Contents field varies widely among record types, but the Record Type field identifies the type of record and how the Contents field is used. Figure 3 lists the formats of all Microsoft object module records.

Listing 1, called WORDC, is a simple Pascal program that counts the number of words in a text file and displays that number on the screen. WORDC was compiled using MS-Pascal version 3.20. The following commands were used to compile the program:

```
PAS1 WORDC.PAS, WORDC.OBJ,
  WORDC.LST, NUL;
PAS2
```

FIGURE 1: *Linking Process*



A translator, such as MS-Pascal, transforms the source code into object code (a). The object code can be added to a library of object modules (b), or it can be used as input to a linker (c). The output from the linker is an executable file (d).

FIGURE 2: *Record Structure*

RECORD TYPE	RECORD LENGTH	CONTENTS	CHECKSUM

The Contents field is the meat of the record. The Record Length field is a two-byte value that specifies the number of bytes in the Contents field. The Checksum field helps the linker or loader identify data errors when reading the module from disk.

The following command links an object code to a Pascal runtime library and produces an executable file:

```
LINK WORDC.OBJ, WORDC.EXE,
  WORDC.MAP, PASCAL.LIB;
```

The object module that results from the compilation of this program (WORDC.OBJ) is shown in figure 4. The left portion of the listing is a hexadecimal dump; the right portion is the corresponding ASCII translation. The start of each record and the record type are noted in the listing.

For clarity, this article refers to each record type by a six-character record name, rather than by a record type number. Intel and Microsoft also use these names to refer to the records in their documentation.

THEADR record (offset 0H). The first record in the object file is a translator header record. It lists the name of the object module without a file extension (in this case, WORDC).

The THEADR record in figure 4 starts at the first byte of the listing (offset 0H). This byte, and any others in the listing, can be found by using the numbers in the OFFSET column (the leftmost column). Each OFFSET entry lists the hexadecimal byte offset of the first byte in that row.

The format of a THEADR record is shown in figure 3-a. The Length and Checksum fields (which every record contains) in this example are structured as described earlier. In the THEADR record in figure 4, the two-byte value listed in the Length field is 07H 00H. This value might be misinterpreted to be 0700H, an awfully large header record. However, word values on Intel-based computers are interpreted with the low-order byte first, followed by the high-order byte. Thus, the length of this record is seven bytes, which defines the length of the Module Name field (a type of Contents field).

The Module Name field in the THEADR record is variable in length; the first byte defines the number of bytes remaining in the Module Name field (a number between 0 and 40), and this number is followed by hexadecimal representations of ASCII characters.

In figure 4, the Module Name field starts with the value 05, indicating that the next five bytes make up the module name. The ASCII characters represented in those bytes are *WORDC*, which is the name of the sample program.

COMENT record (offset 0AH). The next two records are COMENT records, which contain comments that do not affect program execution. COMENT records are created by program translators for use with other utility programs or for personal reference. COMENT records are used more by Intel than by Microsoft development tools. Refer to figure 3-b for the format of a COMENT record.

When the No Purge bit is set to 1, object file utility programs (such as linkers and librarians) should never delete this COMENT record. When the bit is set to 0, however, the utility programs may delete the record. When the No List bit is set to 1, it indicates that utility programs that normally list COMENT records should not list the text of this particular record.

TABLE 1: *Microsoft Object Record Types*

NUMBER (HEX)	TYPE	DESCRIPTION
7A	BLKDEF	Indicates the start of a program block, listing its location and return information.
7C	BLKEND	Indicates the end of a program block.
80	THEADR	Identifies start of modules generated by a translator.
88	COMENT	Denotes a nonexecutable comment.
8A	MODEND	Denotes the end of a module.
8C	EXTDEF	Lists the external variables to which a program refers.
8E	TYPDEF	Describes a variable type.
90	PUBDEF	Defines a program's public symbols.
94	LINNUM	Associates source code line numbers with corresponding object code.
96	LNAMES	Lists names of segments, classes, overlays, and groups.
98	SEGDEF	Describes a program segment.
9A	GRPDEF	Defines the segments that make up a group.
9C	FIXUPP	Identifies places in the data records at which the loader must change references to locations.
A0	LEDATA	Lists logical, enumerated data.
A2	LIDATA	Lists logical, iterated data.

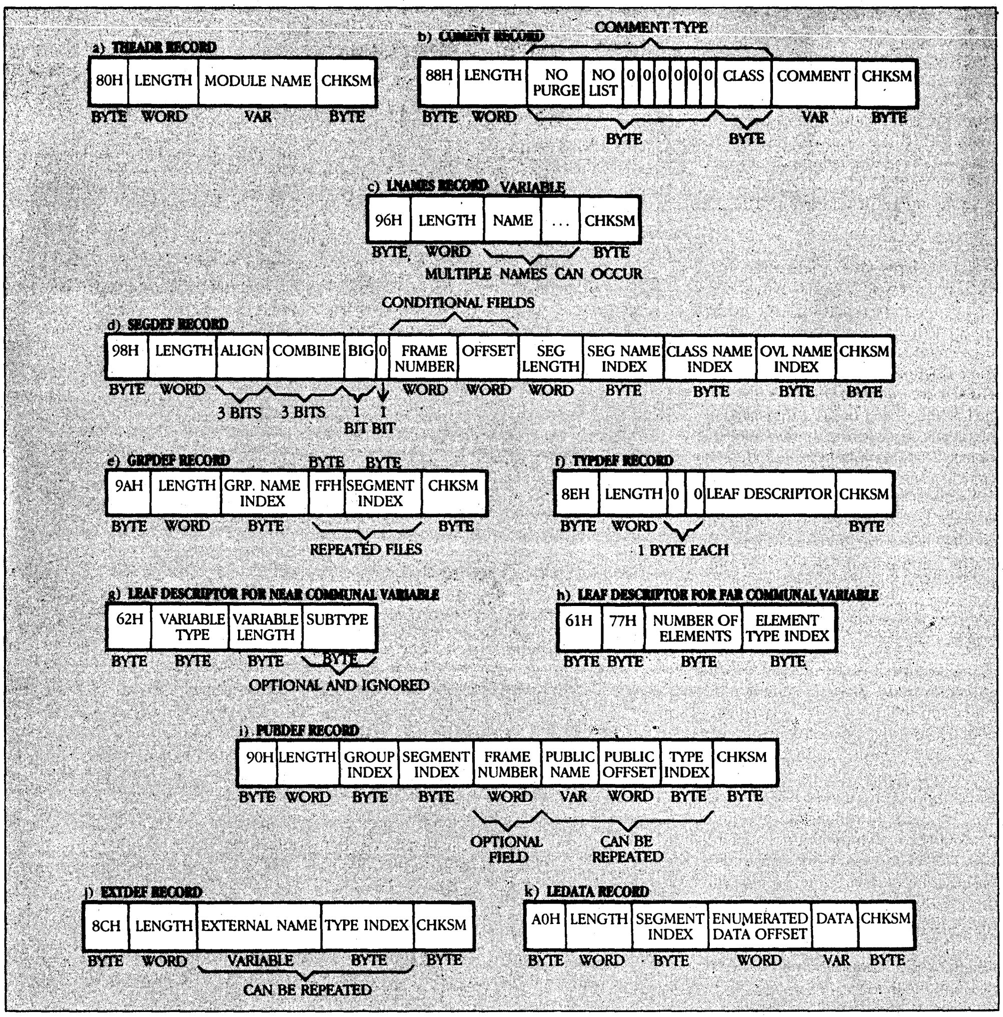
The object module is the unit of information produced by a program translator. Each object module consists of records such as those described here.

TABLE 2: *Additional Intel Record Types*

NUMBER (HEX)	TYPE	DESCRIPTION
6E	RHEADR	Identifies the start of a module that has been processed by Intel's linker. This is a relocatable module that can be loaded directly by Intel's loader.
70	REGINT	Provides information about processor registers.
72	REDATA	Lists relocatable, enumerated data.
74	RIDATA	Lists relocatable, iterated data.
76	OVLDEF	Names and describes a program overlay.
78	ENDREC	Indicates the end of block or overlay.
7E	DEBSYM	Describes the program's local symbols, including stack and based symbols.
82	LHEADR	Identifies the start of a module that has been processed by Intel's linker.
84	PEDATA	Lists physical, enumerated data.
86	PIDATA	Lists physical, iterated data.
92	LOCSYM	Describes the program's local symbols.
A4	LIBHED	Denotes the start of a library file.
A6	LIBNAM	Lists the names of the modules in a library file.
A8	LIBLOC	Lists the locations of all the modules in the library.
AA	LIBDIC	Names all the public symbols in the library.

The complete Intel standard defines 30 different types of records—15 more than Microsoft's standard uses. These additional record types are described here.

FIGURE 3: *Formats of Microsoft Object Records*

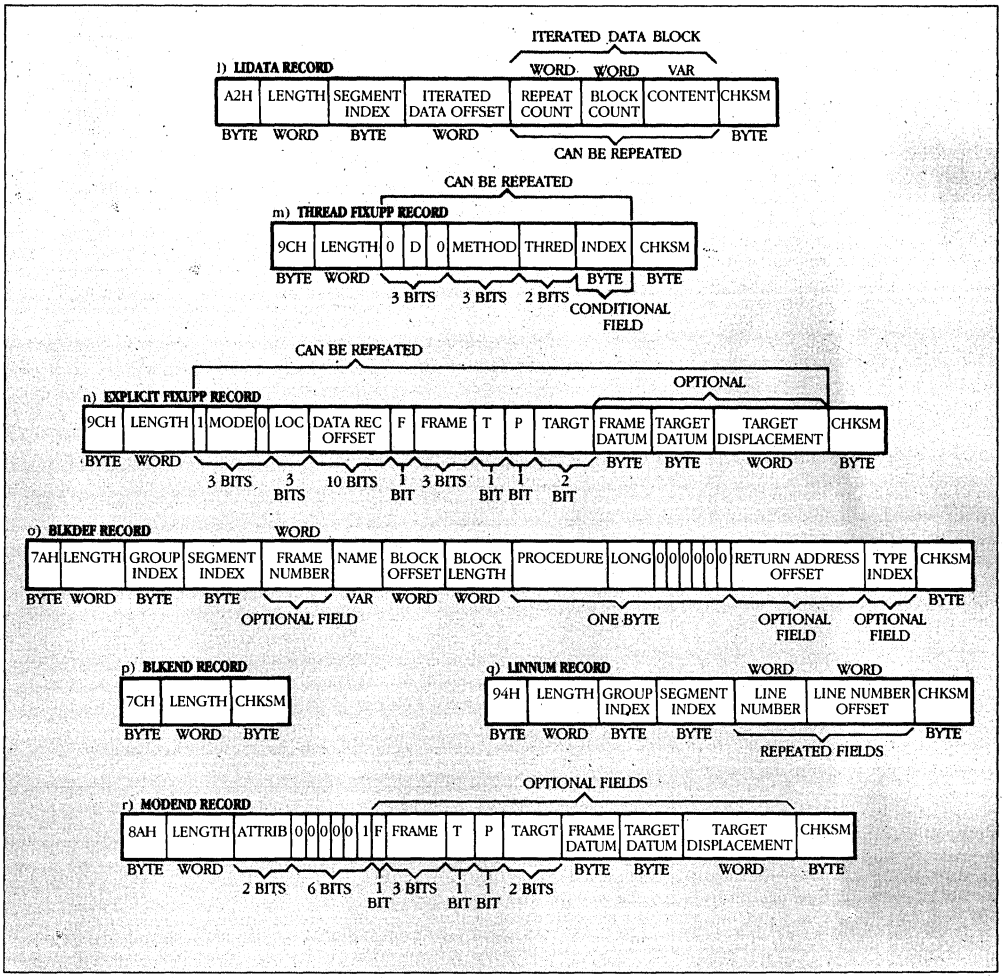

a) THEADR RECORD
80H | LENGTH | MODULE NAME | CHKSM
BYTE WORD VAR BYTE
b) COMENT RECORD
COMMENT TYPE
88H | LENGTH | NO PURGE | NO LIST | 0 | 0 | 0 | 0 | 0 | 0 | CLASS | COMMENT | CHKSM
BYTE WORD BYTE BYTE VAR BYTE
c) LNAMES RECORD VARIABLE
96H | LENGTH | NAME | ... | CHKSM
BYTE WORD BYTE
MULTIPLE NAMES CAN OCCUR
CONDITIONAL FIELDS
d) SEGDEF RECORD
98H | LENGTH | ALIGN | COMBINE | BIG | 0 | FRAME NUMBER | OFFSET | SEG LENGTH | SEG NAME INDEX | CLASS NAME INDEX | OVL NAME INDEX | CHKSM
BYTE WORD WORD WORD WORD BYTE BYTE BYTE BYTE
3 BITS 3 BITS 1 1
BIT BIT
e) GRPDEF RECORD
BYTE BYTE
9AH | LENGTH | GRP. NAME INDEX | FFH | SEGMENT INDEX | CHKSM
BYTE WORD BYTE BYTE
REPEATED FILES
f) TYPDEF RECORD
8EH | LENGTH | 0 | 0 | LEAF DESCRIPTOR | CHKSM
BYTE WORD BYTE
1 BYTE EACH
g) LEAF DESCRIPTOR FOR NEAR COMMUNAL VARIABLE
62H | VARIABLE TYPE | VARIABLE LENGTH | SUBTYPE
BYTE BYTE BYTE BYTE
OPTIONAL AND IGNORED
h) LEAF DESCRIPTOR FOR FAR COMMUNAL VARIABLE
61H | 77H | NUMBER OF ELEMENTS | ELEMENT TYPE INDEX
BYTE BYTE BYTE BYTE
i) PUBDEF RECORD
90H | LENGTH | GROUP INDEX | SEGMENT INDEX | FRAME NUMBER | PUBLIC NAME | PUBLIC OFFSET | TYPE INDEX | CHKSM
BYTE WORD BYTE BYTE WORD VAR WORD BYTE BYTE
OPTIONAL FIELD
CAN BE REPEATED
j) EXTDEF RECORD
8CH | LENGTH | EXTERNAL NAME | TYPE INDEX | CHKSM
BYTE WORD VARIABLE BYTE BYTE
CAN BE REPEATED
k) LEDATA RECORD
A0H | LENGTH | SEGMENT INDEX | ENUMERATED DATA OFFSET | DATA | CHKSM
BYTE WORD BYTE WORD VAR BYTE

The Class field tells which kind of comment this record contains; this varies depending on which translator generated the object code. The field is not important, except that Intel normally reserves values 2-155 for its internal use. Notice that Microsoft uses a restricted value here (81H, or 129 decimal). This probably corresponds to a value that Intel uses in similar circumstances.

The variable-length Comment field contains the text that this record was created to represent. The two comment records in this object module start at bytes 0AH and 19H, respectively. They are used to identify the Microsoft Pascal compiler, as indicated by the text MS PASCAL and PASCAL in the respective Comment fields.

LNAMES record (offset 25H). LNAMES records contain a list of segments, classes, groups, and overlays that are used for reference by other record types. A *segment* is an area of memory that can be accessed with a 16-bit address, at most 64KB long. The starting address of a segment is referenced through one of the segment registers (CS, DS, ES, or SS). A *class* is several segments that are referenced by a common name (the class name). A *group* is one or more segments that fit into a single 64KB area of memory, can be referenced by a common name (the group name), and share the same segment register value. *Overlays* are separate sections of code that share a given memory area in order to optimize program size. Overlays sharing the same memory area cannot execute simultaneously.

Instead of duplicating the complete names of segments, classes, groups, and overlays in record after record, other object module records refer to these names by a number that indicates the



l) LIDATA RECORD
ITERATED DATA BLOCK
A2H | LENGTH | SEGMENT INDEX | ITERATED DATA OFFSET | REPEAT COUNT | BLOCK COUNT | CONTENT | CHKSM
BYTE WORD BYTE WORD WORD WORD VAR BYTE
CAN BE REPEATED
m) THREAD FIXUPP RECORD
CAN BE REPEATED
9CH | LENGTH | 0 | D | 0 | METHOD | THRED | INDEX | CHKSM
BYTE WORD 3 BITS 3 BITS 2 BITS BYTE BYTE
CONDITIONAL FIELD
n) EXPLICIT FIXUPP RECORD
CAN BE REPEATED
OPTIONAL
9CH | LENGTH | 1 | MODE | 0 | LOC | DATA REC OFFSET | F | FRAME | T | P | TARGT | FRAME DATUM | TARGET DATUM | TARGET DISPLACEMENT | CHKSM
BYTE WORD 3 BITS 3 BITS 10 BITS 1 BIT 3 BITS 1 BIT 1 BIT 2 BIT BYTE BYTE WORD BYTE
o) BLKDEF RECORD
7AH | LENGTH | GROUP INDEX | SEGMENT INDEX | FRAME NUMBER | NAME | BLOCK OFFSET | BLOCK LENGTH | PROCEDURE | LONG | 0 | 0 | 0 | 0 | 0 | 0 | RETURN ADDRESS OFFSET | TYPE INDEX | CHKSM
BYTE WORD BYTE BYTE WORD VAR WORD WORD BYTE
OPTIONAL FIELD
ONE BYTE
OPTIONAL FIELD
OPTIONAL FIELD
p) BLKEND RECORD
7CH | LENGTH | CHKSM
BYTE WORD BYTE
q) LINNUM RECORD
94H | LENGTH | GROUP INDEX | SEGMENT INDEX | LINE NUMBER | LINE NUMBER OFFSET | CHKSM
BYTE WORD BYTE BYTE WORD WORD BYTE
REPEATED FIELDS
r) MODEND RECORD
OPTIONAL FIELDS
8AH | LENGTH | ATTRIB | 0 | 0 | 0 | 0 | 0 | 1 | F | FRAME | T | P | TARGT | FRAME DATUM | TARGET DATUM | TARGET DISPLACEMENT | CHKSM
BYTE WORD 2 BITS 6 BITS 1 BIT 3 BITS 1 BIT 1 BIT 2 BITS BYTE BYTE WORD BYTE

The formats of all the Microsoft object records are shown here. The complete Intel standard offers another 15 record types.

order in which the names appear in the LNAMES records. Number 1 is the first name in the first LNAMES record; the last number is the last name in the last LNAMES record. The format of an LNAMES record is shown in figure 3-c.

Each Name field consists of a one-byte length followed by the characters that specify the name of a segment, class, group, or overlay. As many name fields as necessary are allowed, with each one having the same structure.

The LNAMES record contained in figure 4 starts at byte 25H. It has a length of 44H bytes. The first name in the record is HIMEM and is five bytes long. The other names are LARGE, CONST, COMADS, DATA, STACK, MEMORY, HEAP, CODE, WORDC, a null name (as indicated by a length of 0), and DGROUP. Of all these names, only WORDC (the name of the program) is defined by the program shown in listing 1; the others are internally generated by the Microsoft Pascal compiler.

SEGDEF record (offset 6CH). SEGDEF (segment definition) records describe the segments that make up the program. Figure 3-d shows the format of a SEGDEF record. The first field after the standard Length field is Align, a three-bit field that specifies the alignment of the segment. The possible values in this field are described in table 3. Another three-bit field, the Combine field, describes the rules the linker must follow in order to combine this segment with others of the same name. This field has the values shown in table 4.

The Big field is a one-bit field that, if set to 1, indicates that the segment is exactly 64KB (65,536 bytes) long (note that a segment cannot be greater than 64KB in length). If the segment is less than 64KB long, this field is 0.

The Frame Num and Offset fields are present only for absolute segments (Align field contains 0). They identify the starting address of the absolute segment. The Frame Num field specifies the number of the frame containing the segment. A *frame* is a type of segment, using Intel terminology. It is a 64KB area of address space that begins on an even, 16-byte boundary (Intel defines this as a paragraph boundary). The frame beginning at address 0 is frame 0, the one beginning at address 10H (16 decimal) is frame 1, etc. The Offset is the number of bytes from the start of the frame at which the segment begins.

The Seg Length field is a word that lists the length of the segment in bytes. Because the largest value that a word can contain is 65,535, if the segment is exactly 64KB (65,535 bytes) long, this field is set to 0 and the Big field (described earlier) is set to 1.

The Seg Name Index, Class Name Index, and Ovl Name Index fields are single-byte fields that identify the segment's name, class, and overlay by specifying the numbers of the appropriate names in the LNAMES record. For example, a 2 indicates the second name in the LNAMES record, a 3 indicates the third name in the record, etc.

An examination of the first SEGDEF record in figure 4 will help to explain how SEGDEF records work. This record starts at byte 6CH. The first byte in the record (98H) identifies the record as a SEGDEF, while the next two bytes, 07H 00H, indicate the length of the rest of the record (7 bytes).

The next byte, 40H, represents the Align, Combine, and Big fields. To pick out the individual fields, 40H must be translated into its binary equivalent: 010 000 00. The leftmost three bits, binary 010 (or 2, in decimal form), give the value for the Align field. This indicates that the segment is a relocatable, word-aligned segment.

The rest of the bits are 0; therefore, the segment is a private segment (Combine field is 000) less than 64KB long (Big field is 0). Because the segment is not an absolute segment, the Frame and Offset fields are absent. The next word, EFH 00H, gives the length of the segment (239 decimal bytes).

The next three bytes (0AH, 09H, and 0BH) indicate the names of the segment, the class, and the overlay, respectively. These values are indices into the list of names in the LNAMES record. To interpret the names, go back to the LNAMES record (it starts at byte 25H in figure 4) and count to the tenth, ninth, and eleventh names. Counting to the tenth name reveals the segment name to be WORDC. The ninth name (the class name) is CODE, and the eleventh name, the overlay name, is null (this program has no overlays).

Examining the rest of the SEGDEF records (there are seven of them in all) shows that the program contains segments named WORDC, HEAP, STACK, DATA, COMADS, CONST, and HIMEM.

GRPDEF record (offset B2H). The record that follows the SEGDEF records is a GRPDEF (group definition) record. This record defines a group by naming it and by identifying all the segments that make up the group. (A group consists of several segments that fit into one 64KB area of memory. When these segments are combined as a group, one segment register can access any one of the individual segments.)

Figure 3-e illustrates the format of a GRPDEF record. In this record, the Grp Name Index, much like the other indexes that were described previously in this article is an index that specifies which of the names in the LNAMES record are used to identify the group.

The Segment Index identifies a segment that helps make up the group. It is a byte whose value is an index into the list of SEGDEF records discussed earlier. For example, a 1 indicates the first SEGDEF record, a 2 indicates the second SEGDEF record, and so on. As many Segment Indexes as necessary are allowed and each is preceded by the hexadecimal value FFH.

The GRPDEF record in figure 4 starts at byte B2H. It is 12 bytes long (0CH 00H), and its name is DGROUP (the twelfth name in the LNAMES list). This group has five segments, referencing the sixth, fifth, fourth, third, and second SEGDEF records in the module (HEAP, STACK, DATA, COMADS, and

TABLE 3: *Possible Align Values*

VALUE	MEANING
0	Absolute segment. In this case, both the Frame and Offset fields are present to describe the location of the segment.
1	Relocatable and byte-aligned segment (the segment can begin at any memory address).
2	Relocatable and word-aligned segment (the segment must begin at an address on a word boundary).
3	Relocatable and paragraph-aligned segment (the segment must begin at an address on a 16-byte boundary).
4	Relocatable and page-aligned segment (the segment must begin at an address on a 256-byte boundary).

The Align field is a three-bit field that specifies the alignment of the segment. The possible values in this field are listed and described here.

TABLE 4: *Possible Combine Values*

VALUE	MEANING
0	A private segment that cannot be combined, even with other segments of the same name, and that has its own unique base address.
2	A public segment. All data in segments of the same name and class are loaded into sequential memory locations as one segment and can be referenced with a single segment-register value.
5	A stack segment. All data in segments of the same name and class are loaded into sequential memory locations as one segment and can be referenced with a single segment-register value. The LINK program places the segment address into the SS register (the SP register contains the Offset field value if provided; otherwise, it contains no determined value) when an .EXE file is loaded for execution.
6	A common segment. All segments of the same name and class are loaded overlapped into a single segment (each module's segment reference begins at a memory address with offset 0). The size of the common segment is the size of the largest segment definition.

The Combine field, which can have the values listed above, describes the rules the linker must follow to combine the Align segment with others of the same name.

CONST). (This is the normal DGROUP produced by the MS-Pascal compiler. MS-Pascal loads the segment address for DGROUP into the DS register during the program's initialization.)

TYPDEF record (offset C1H). The next record is a type definition record. In the complete Intel format, TYPDEF records contain a wealth of information about public variables, external variables, code blocks, debugging symbols, and local symbols. These records do not affect program execution, but they do provide useful information to high-level language debuggers.

Microsoft translators, however, usually generate just a single dummy TYPDEF that is used as a placeholder reference for fields in other object module records. (Some other records need to reference a TYPDEF, and those records can always refer back to this dummy TYPDEF.) In cases in which a null TYPEDEF is used, almost all TYPDEFs can be ignored.

The one time TYPDEF records do contain meaningful information is when communal variables are involved. A *communal variable* is an uninitialized public variable whose size is not fixed at compilation time (such as a FORTRAN common block). With communal variables, the same variable might be defined with different size declarations in several different modules. (The only PC language that uses communal variables is FORTRAN.)

TYPDEF records appear for communal variables, describing the type and size of each. The linker then allocates space for the largest such variable defined, and all references to a given communal variable have an identical starting address and varying lengths.

The format of a TYPDEF record is shown in figure 3-f. In this record, the Leaf Descriptor has one of two formats, depending on whether the communal variable is a *near variable* (its address is described only with a 16-bit offset value) or a *far variable* (its 32-bit address contains both segment and offset values). For near variables, the first byte of the Leaf Descriptor is 62H (see figure 3-g). The Variable Type field is a byte that describes the type of the communal variable with the following values:

77H Array
79H Structure
7BH Scalar

The Variable Length field lists the length of the variable in bits. The Subtype, in the full Intel format, then specifies additional information about the variable. In the Microsoft format, this field has no meaning and, therefore, usually is not present.

If the communal variable is a far variable (figure 3-h), the first byte of the Leaf Descriptor field has a value of 61H. The next byte, 77H, means the variable is an array (see the list of variable types above). The *MS-DOS Programmer's Reference* states that arrays are the only far variables to appear.

The Number of Elements field lists the number of entries in the array. The Element Type Index is an index into the list of previously defined TYPDEF records. For example, a 2 would indicate the second TYPDEF record; the TYPDEF pointed to is a TYPDEF for a near variable, and it identifies the type of the array elements.

Examining the TYPDEF record shown in figure 4 (beginning at byte C1H) illustrates that this TYPDEF is simply a dummy record and can be ignored. The length (05H 00H) is five bytes. The next two bytes (00H 00H) are the 0 bytes shown in the TYPDEF format. The next byte (7BH) indicates that this variable is a scalar. The 0 byte that follows indicates that the length of the scalar is 0. Therefore, this TYPDEF record does not provide any real information; it merely defines a variable with length 0.

In Microsoft object records, the Type Index field refers to the dummy TYPDEF, because Microsoft languages do not generate TYPDEF records.

PUBDEF record (offset C9H). Next comes a PUBDEF (public names definition) record, which describes all the public symbols in this object module. *Public symbols* are names of variables, procedures, and functions that are defined in this module and that can be referenced by other modules. PUBDEF records provide information about these symbols in order that the linker can match them with external symbols that appear in other modules.

PUBDEF records list the names of public symbols, the group and segment in which they reside, and their offset from the start of the segment. Figure 3-i shows the format of the PUBDEF record. The Group Index is an index into the LNAMES record that identifies the name of the group associated with this public symbol. A value of 0 indicates that the symbol has no associated group. The Segment Index identifies the segment containing this symbol by referencing a SEGDEF record (a 1 indicates the first SEGDEF, a 2 indicates the second, and so on).

If both the Group Index and the Segment Index are 0, the Frame Number field is present; otherwise, the Frame Number field does not appear. If present, the Frame Number lists the number of a frame containing the public symbol. (Remember, a frame is a 64KB block of memory that starts on a 16-byte boundary. Frame 0 goes from 0 to 64K; frame 1 goes from 10H to 64K+10H; and so on.)

The Public Name, Public Offset, and Type Index fields are repeated for each public name defined in this record. The Public Name field lists the name of the public symbol; the first byte lists the number of bytes in the name, followed by an ASCII representation of the name. The Public Offset field (a word value) specifies the number of bytes (the offest) from the start of the segment or frame containing the symbol.

The Type Index field identifies the TYPDEF record that describes this symbol. Like the Segment Index field this field lists the number of a previously defined TYPDEF record (a 1 indicates the first TYPDEF, a 2 indicates the second, etc.). In Microsoft object records, this field refers back to the dummy TYPDEF, because Microsoft languages do not generate TYPDEF records.

The sample program shown in listing 1 did not explicitly define any public symbols, but the compiler generated two of them. An examination of the PUBDEF record in figure 4 follows, beginning at byte C9H.

The record length (16H 00H) is 22 bytes. The Group Index is 00H (no group is associated with these symbols). The Segment Index is 01H, indicating that the symbols are contained in the first segment that is defined in a SEGDEF record (WORDC).

The name of the first symbol is next. It is five bytes long and is called WORDC (the program name). Its offset is 1 (01H 00H), and its Type Index is also 1 (there is only one TYPDEF).

The next public symbol is six bytes long and has the name ENTGQQ. It also has an offset of 1 as well as a Type Index of 1. This means that the compiler has defined two different names for the same symbol. ENTGQQ is used by the Pascal runtime library as a common method for identifying the beginning of the main program. (Any symbols that end with the letters *QQ* most likely have been generated by MS-Pascal.)

EXTDEF record (offset E2H). The EXTDEF record, which appears next in figure 4, defines the names of external symbols to which this program refers. *External symbols* are the names of variables or subroutines that a program uses but does not define (they are defined in other object modules and declared public there). Figure 3-j illustrates the format of an EXTDEF record.

The sample program declared no external subroutines or variables, but the compiler generated external references to some of its library routines. The EXTDEF record beginning at byte E2H of figure 4 lists them.

The first word (71H 00H) indicates the length of the record (113 bytes). The remaining portion of the record names the externally defined variables and identifies their TYPDEF records. This record lists symbols named BEGXQQ, WTLFQQ, WTIFQQ, WTSFQQ, RTCFQQ,EOFFQQ, PPEFQQ, RTAFQQ, PPMFQQ, NEWFQQ, INIFQQ, INPFQQ, OUTFQQ, and RESFQQ, all referencing TYPEDEF record 1 (which indicates a null record).

These external variables are defined in the Pascal runtime library. Because the compiler always generates its own external references, programmers must always link their Pascal programs to the Pascal libraries that accompany the MS-Pascal compiler.

LEDATA and LIDATA records (offset 164H). The next record in figure 4 is a FIXUPP record, which identifies locations in the data that must be changed before a program can be loaded into memory. To understand FIXUPP records, the user must know how data records work.

Data records contain the actual machine code and data that get loaded into memory. Two kinds of data records can appear in a Microsoft object module: LEDATA (logical enumerated data) and LIDATA (logical iterated data). An LEDATA record contains data just as they will appear when loaded into memory (except where they will be modified by FIXUPP records). The LIDATA record is encoded to compress repeated data, thereby making the object module smaller.

The format of an LEDATA record is shown in figure 3-k. In this record, both the Segment Index and Enumerated Data Offset fields help identify where the data will reside in memory. The Segment Index is the number of a previously defined SEGDEF record. The Enumerated Data Offset is a word that indicates where the data begin, relative to the start of that segment. The variable-length DATA field gets loaded into memory at the address specified by the Segment Index and Data Offset.

The first LEDATA record in listing 2 starts at byte 164H. This record is eight bytes long (08H 00H). The Segment Index (04H) identifies the fourth SEGDEF (DATA) as the segment containing this data. The address offset from the start of that segment is 0 (00H 00H). The remaining bytes of the record are the data that get loaded into the DATA segment. Other LEDATA records in this module can be interpreted in the same way.

Several LEDATA records are listed for this sample program. The largest of the records contains the program's code, which will reside in the code segment. Each LEDATA record can contain only 1,024 bytes of information; thus, if the sample program's code were longer, additional LEDATA records would be present. Other LEDATA records designate information for other segments, including program data such as the text *word count* =, which is placed into the CONST segment. (Listing 1 shows that these are words that the program displays on the screen.)

Data blocks can be nested as many as 17 levels deep, but the size of the iterated data block field cannot exceed 512 bytes.

The term *logical enumerated data* implies two characteristics of the data in the record. First, the data are logical. In Intel terminology, that means relocatable; the loader is free to load the data wherever necessary, based on available system memory and the software that is already executing.

Second, the data are enumerated, which means that every byte is listed, even if many consecutive bytes contain the same value. Another type of data (iterated) is encoded to decrease the size of the data record. Iterated data records compress consecutive bytes that contain the same value. Although the sample program contains no iterated data records, programs that load a lot of repeated data might.

The Segment Index and Iterated Data Offset fields contained in the LIDATA record are exactly the same as the similarly named fields in the LEDATA record (refer to figure 3-l); they identify an address for loading data. The next three fields in LIDATA (Repeat Count, Block Count, and Content) are called an *iterated data block* and specify the iterated data. Repeat Count is a word that indicates the number of times the Content field is to be repeated. If the Repeat Count is 3, three copies of the Content field will be loaded into memory.

The next field, Block Count, is a word that identifies whether the Content field contains only data or whether the Content field itself is made up of iterated data blocks (Repeat Count, Block Count, and Content fields). If the Block Count field is 0, the Content field contains only data, which are loaded into memory as many times as indicated by the Repeat Count field. However, if the Block Count field is not 0, the Content field itself contains iterated data blocks; that is, the iterated data blocks are nested. The number in the Block Count field specifies the number of iterated data blocks in the Content field.

If the Content field contains only data, the first byte indicates the number of bytes of data in the rest of the field. However, if the Content field does not contain data, it is interpreted as another iterated data block, with the first word being a Repeat Count.

Figure 5 illustrates how the iterated data block can be nested to compress the repeated data. The first part of the figure shows a simple iterated data block with a single Repeat Count and Block Count. The second part of the figure illustrates an iterated data block with three levels of nesting. This nesting of data blocks can go as many as 17 levels deep, as long as the size of the iterated data block field does not exceed 512 bytes (this is a limitation of the Microsoft linker).

If a program loads a great deal of repeated data (such as defining a large array initialized to a common value), LIDATA records allow that repeated data be stored in very few bytes. Of course, LIDATA records are not appropriate when no repeated data are present. Compilers and assemblers are free to decide which record to use; they may determine which will cause the resulting object file to be smaller.

FIXUPP record (offset 156H). FIXUPP records identify data in either LEDATA or LIDATA records that refer to symbols whose locations will change as a result of the linker deciding how to relocate everything. FIXUPP records are notes to

FIGURE 4: *Object Module from WORDC.OBJ*

```
Hexidecimal Dump of File: b:wordc.obj

OFFSET                   HEX                              ASCII

       \/·THEADR                     \/·COMENT
0000   80 07 00 05 57 4F 52 44 43 F5 88 0C 00 00 00 4D   ....WORDC......M

                                  \/·COMENT
0010   53 20 50 41 53 43 41 4C F8 88 09 00 00 81 50 41   S PASCAL......PA

                      \/·LNAMES
0020   53 43 41 4C 3A 96 44 00 05 48 49 4D 45 4D 05 4C   SCAL:.D..HIMEM.L
0030   41 52 47 45 05 43 4F 4E 53 54 06 43 4F 4D 41 44   ARGE.CONST.COMAD
0040   53 04 44 41 54 41 05 53 54 41 43 4B 06 4D 45 4D   S.DATA.STACK.MEM
0050   4F 52 59 04 48 45 41 50 04 43 4F 44 45 05 57 4F   ORY.HEAP.CODE.WO

                                           \/·SEGDEF
0060   52 44 43 00 06 44 47 52 4F 55 50 E4 98 07 00 40   RDC..DGROUP....@

                         \/·SEGDEF
0070   EF 00 0A 09 0B 14 98 07 00 44 00 00 08 07 0B 03   .........D......

       \/·SEGDEF                     \/·SEGDEF
0080   98 07 00 54 00 00 06 06 08 F6 98 07 00 48 12 05   ...T.........H..

                   \/·SEGDEF                     \/·SEGDEF
0090   05 05 0B ED 98 07 00 48 00 00 04 04 0B 06 98 07   .......H........

                               \/·SEGDEF
00A0   00 48 1C 00 03 03 08 EC 98 07 00 48 00 00 01 01   .H.........H....

             \/·GRPDEF
00B0   0B 0C 9A 0C 00 0C FF 06 FF 05 FF 04 FF 03 FF 02   ................

          \/·TYPDEF               \/·PUBDEF
00C0   3F 8E 05 00 00 00 7B 00 F2 90 16 00 00 01 05 57   ?.....{........W
00D0   4F 52 44 43 01 00 01 06 45 4E 54 47 51 51 01 00   ORDC....ENTGQQ..

             \/·EXTDEF
00E0   01 F8 8C 71 00 06 42 45 47 58 51 51 01 06 57 54   ...q..BEGXQQ..WT
00F0   4C 46 51 51 01 06 57 54 49 46 51 51 01 06 57 54   LFQQ..WTIFQQ..WT
0100   53 46 51 51 01 06 52 54 43 46 51 51 01 06 45 4F   SFQQ..RTCFQQ..EO
0110   46 46 51 51 01 06 50 50 45 46 51 51 01 06 52 54   FFQQ..PPEFQQ..RT
0120   41 46 51 51 01 06 50 50 4D 46 51 51 01 06 4E 45   AFQQ..PPMFQQ..NE
0130   57 46 51 51 01 06 49 4E 49 46 51 51 01 06 49 4E   WFQQ..INIFQQ..IN
0140   50 46 51 51 01 06 4F 55 54 46 51 51 01 06 52 45   PFQQ..OUTFQQ..RE

                         \/·FIXUPP
0150   53 46 51 51 01 19 9C 08 00 00 06 01 04 02 02 03   SFQQ............

                   \/·LEDATA                        \/·LEDATA
0160   01 44 01 01 A0 08 00 04 00 00 42 57 02 00 B9 A0   .D........BW....
0170   10 00 04 05 00 57 4F 52 44 43 2E 50 41 53 20 20   .....WORDC.PAS

             \/·LEDATA               \/·LEDATA
0180   20 56 A0 05 00 04 04 00 0C 47 A0 10 00 06 02 00    V.......G......

                                              \/·LEDATA
0190   07 49 4E 5F 46 49 4C 45 4C 41 4C 00 52 A0 3F 00   .IN_FILELAL.R.?.
01A0   01 01 00 55 B8 EC 81 EC 04 00 9A 00 00 00 00 B8   ...U............
01B0   90 02 50 B8 50 00 50 B8 01 00 50 9A 00 00 00 00   ..P.P.P...P.....
01C0   B8 14 00 50 B8 50 00 50 B8 01 00 50 9A 00 00 00   ...P.P.P...P....
                                                    \/·FIXUPP
01D0   00 B8 07 00 50 B8 03 00 50 9A 00 00 00 00 E8 9C   ....P...P.......
01E0   1A 00 CC 08 56 08 C4 00 80 CC 19 56 0A C4 1E 80   ....V......V....

                                           \/·LEDATA
01F0   CC 2A 56 0A C4 33 8C CC 37 56 09 C8 A0 12 00 06   .*V..3..7V......
0200   0E 00 00 77 6F 72 64 20 63 6F 75 6E 74 20 3D 20   ...word count =

          \/·LEDATA
0210   AB A0 B7 00 01 3C 00 B8 00 00 50 B8 14 00 50 9A   .....<....P...P.
0220   00 00 00 00 9A 00 00 00 00 B8 14 00 50 9A 00 00   ............P...
0230   00 00 C7 06 0C 05 00 00 C6 06 10 05 00 B8 14 00   ................
0240   50 9A 00 00 00 00 D1 E8 72 4A B8 14 00 50 B8 0E   P.......rJ...P..
0250   05 1E 50 33 C0 50 B8 FF 00 50 9A 00 00 00 00 80   ..P3.P...P......
0260   3E 0E 05 20 74 15 80 3E 0E 05 09 74 0E 80 3E 0E   >.. t..>...t..>.
0270   05 00 74 07 80 3E 0E 05 0A 75 07 C6 06 10 05 00   ..t..>...u......
0280   EB 10 F6 06 10 05 01 75 09 C6 06 10 05 01 FF 06   .......u........
0290   0C 05 EB A9 B8 00 00 50 B8 00 00 50 B8 0F 00 1E   .......P...P....
02A0   50 B8 FF 7F 50 50 9A 00 00 00 00 B8 00 00 50 FF   P...PP........P.
02B0   36 0C 05 B8 FF 7F 50 50 9A 00 00 00 00 B8 00 00   6.....PP........

                                        \/·FIXUPP
02C0   50 9A 00 00 00 00 8B E5 5D CB 03 9C 64 00 C4 01   P.......]...d...
02D0   86 0C C4 05 8D CC 09 56 08 CC 0E 56 07 C4 13 8D   .......V...V....
02E0   CC 17 56 0E C4 1D 8D C4 23 8D C4 27 8D CC 2B 56   ..V.....#..'..+V
02F0   06 C4 34 8D C4 38 8D CC 44 56 05 C4 4A 8D C4 51   ..4..8..DV..J..Q
0300   8D C4 58 8D C4 5F 8D C4 66 8D C4 6D 8D C4 74 8D   ..X.._..f..m..t.
0310   C4 79 8D C4 7E 86 00 C4 86 8C CC 90 56 04 C4 95   .y..~.......V...
0320   86 00 C4 9A 8D CC A2 56 03 C4 A7 86 00 CC AB 56   .......V.......V

             \/·BLKDEF
0330   02 B2 7A 11 00 00 01 05 57 4F 52 44 43 01 00 EE   ..z.....WORDC...

                         \/·BLKEND   \/·LINNUM
0340   00 60 00 00 00 A1 7C 01 00 83 94 3B 00 00 01 11   .'....|....;....
0350   00 4E 00 12 00 57 00 13 00 5D 00 14 00 62 00 16   .N...W...]...b..
0360   00 6F 00 17 00 84 00 18 00 A0 00 1C 00 A7 00 1E   .o..............
0370   00 AE 00 1F 00 B3 00 20 00 B7 00 21 00 B7 00 22   ....... ...!..."

                               \/·MODEND
0380   00 B9 00 23 00 EB 00 AE 8A 02 00 00 74            ...#........t
```

The first byte of each record comprising WORDC.OBJ is marked with a *V* symbol, with the record type written beside it.

the linker that essentially say, "Here is a place (called a *location*) where the code refers to a symbol (called a *target*). If the target is moved around, the location's reference in the code should be changed accordingly."

FIXUPP records supply several pieces of information that the linker needs to fix up a location's reference. Figure 6 illustrates such information. First, the FIXUPP record identifies the location to be fixed up. It specifies this as an offset into the data portion of the data record (a FIXUPP always applies to the closest previous LEDATA or LIDATA record). Next, the FIXUPP record identifies the target—that is, the symbol being referenced by the location.

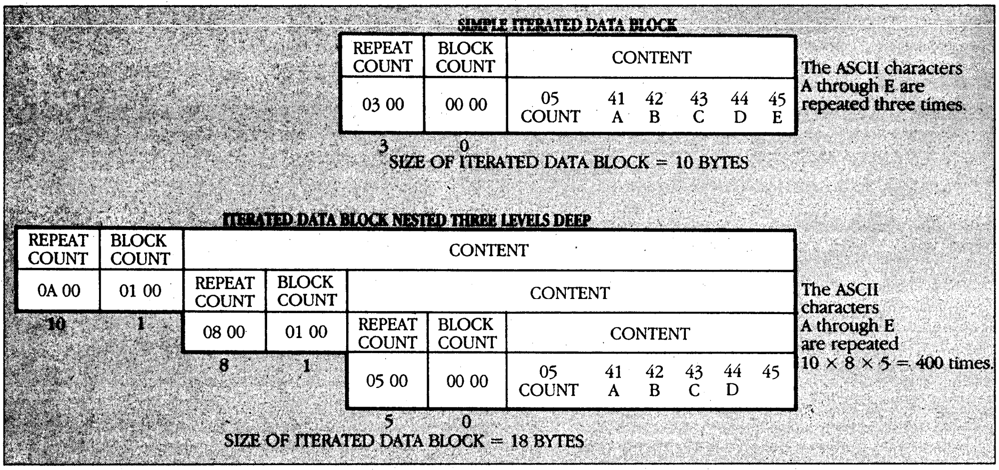
Once the target and location are identified, the FIXUPP record provides information about the framework of the fix-up, that is, how to change the location so that it accurately refers to the target. There are two such pieces of information: the *mode* and the *frame*.

The mode tells the linker whether to change the location's reference to the target using a value relative to the location itself (self-relative mode) or to make the change in the target reference relative to the start of some segment (segment-relative mode). In self-relative mode, the location and target reside in the same segment, and the linker is required to change only the offset portion of the location's reference to the target. In segment-relative mode, the linker needs to change both the base and the offset reference to the target.

The frame is a 64KB area of memory starting on a 16-byte boundary. It provides the information the linker needs to change the base portion of the target's address. For self-relative references, this frame is usually the segment referenced in the address of the location and target. For segment-relative references, the frame is the segment referenced in the address of the target.

Knowing the mode and the frame, the linker is able to change any location's reference accurately, whether or not the target resides in the same segment as the location.

FIGURE 5: *Iterated Data Block*


SIMPLE ITERATED DATA BLOCK
REPEAT COUNT
BLOCK COUNT
CONTENT
03 00
00 00
05 COUNT
41 A
42 B
43 C
44 D
45 E
The ASCII characters A through E are repeated three times.
3
0
SIZE OF ITERATED DATA BLOCK = 10 BYTES
ITERATED DATA BLOCK NESTED THREE LEVELS DEEP
REPEAT COUNT
BLOCK COUNT
CONTENT
0A 00
01 00
10
1
REPEAT COUNT
BLOCK COUNT
CONTENT
08 00
01 00
8
1
REPEAT COUNT
BLOCK COUNT
CONTENT
05 00
00 00
05 COUNT
41 A
42 B
43 C
44 D
45
The ASCII characters A through E are repeated 10 × 8 × 5 = 400 times.
5
0
SIZE OF ITERATED DATA BLOCK = 18 BYTES

The iterated data block can be nested to compress repeated data. In the LIDATA record, this block can be nested 17 levels deep.

FIXUPP records are of two types: *thread* and *explicit fix-up*. A thread FIXUPP record defines locations in memory and the symbols to which they refer. Explicit FIXUPP records reference threads in the same manner as SEGDEF records establish the relationship between a location, target, and frame. By defining commonly used information only once, thread records allow explicit FIXUPP records to be shorter. The format of thread and explicit FIXUPP records is shown in figures 3-m and 3-n.

In a thread FIXUPP, the D field is a single bit that indicates the kind of thread in question. If D is 0, the thread is a target thread, and the information supplied here is used to identify the target of the fix-up (the symbol to which the location in memory refers). If D is 1, the thread is a frame thread, which supplies information about the frame (a segment, a 64KB area of memory starting on a 16-byte boundary). The linker needs to know the frame in order that it can correctly adjust the location's reference to the target.

The Method field is a three-bit field that lists the method the linker must use to identify the target or the frame, based on the value of the D field. If D=0 (a target thread), Method can have one of the values shown in table 5. If D=1 (a frame thread), Method can have one of the values shown in table 6.

The Thread field is a two-bit field that assigns a number (from 0 to 3) to the thread being defined. Later, FIXUPP records can refer to targets or frames using these numbers, instead of having to define the target or frame explicitly. A single thread FIXUPP record can define up to four target threads and four frame threads. If a later thread FIXUPP record assigns the same number to a new thread, it redefines that thread number for remaining FIXUPP records.

The Index field identifies the SEGDEF, GRPDEF, or EXTDEF index referred to by the Method field. For example, if this is a target thread that uses method T0 (see table 5), the Index field is an index into the list of SEGDEF records (a 1 indicates the first SEGDEF defined, a 2 indicates the second SEGDEF, and so on). Likewise, if this thread were a frame thread that used method F2, the Index field would be an index into the EXTDEF list of external symbols.

The Index field always appears except for frame threads that use methods F3, F4, or F5. In those cases, the Index field is unnecessary. The combination of D, Method, Thread, and Index fields defines a single thread. Each thread FIXUPP can define four target threads and four frame threads.

An examination of one of the threads in the first FIXUPP record in figure 4 (starting at byte 156H) will help explain how thread FIXUPPs work. The length of the record (0BH 00H) is 11 decimal bytes. The next byte (00H) contains the D, Method, and Thread fields for the first thread. Because the byte is all 0s, the hexadecimal value does not need to be expanded into binary. It is easy to see that D=0 (this is a target thread), Method is T0 (the target is specified by a SEGDEF index and an offset), and the thread number is 0.

The next byte (06H) is an index that identifies this target. Because the Method field indicates a SEGDEF index, the 06H value is an index into the list of SEGDEF records. It identifies the sixth SEGDEF defined (the CONST segment). When later FIXUPP records refer to target thread 0, they will be referring to the CONST segment.

The other four threads in this record can be interpreted in the same way. Target thread 1 refers to segment 4 (the DATA segment). Target thread 2 refers to segment 2 (the HEAP segment). Target thread 3 refers to segment 1 (the WORDC segment). Frame thread 0 refers to group 1 (DGROUP).

A thread FIXUPP record merely provides common information to be used by other explicit FIXUPP records. An explicit FIXUPP record identifies the location that is to be fixed up, the symbol (or target) to which the location refers, and the context (or frame) in which this fix-up is to take place. Using this information, the linker is then able to adjust the addresses.

In the explicit FIXUPP record in figure 3-n, the Mode field is a single bit that indicates whether the fix-up is self-relative (mode 0) or segment-relative (mode 1). A self-relative FIXUPP lets the linker know that it needs to support 8- and 16-bit offsets without segment values (because the program uses only call, jump, and short jump instructions).

Segment-relative FIXUPPs, however, require, the linker to support all 8088 addressing modes.

The Loc field is a three-bit field that identifies the kind of location to be fixed up. The possible values for this field are shown in table 7.

The Data Rec Offset field is a 10-bit field that identifies the start of the location to be fixed up. It is the offset from the start of the data portion of the preceding data record (either LEDATA or LIDATA). For example, if the Loc field identified the location as a *lobyte* and the Data Rec Offset were 2, the FIXUPP record would tell the loader to change the address of the second byte in the previous data record. Instructions for changing this byte are supplied later in the FIXUPP record.

The F field in the FIXUPP record is a single bit that indicates how the frame for this fix-up is specified. The frame is the context in which the fix-up occurs; that is, it idenifies a 64KB block of memory address space, starting on a 16-byte boundary, that will eventually contain the target as soon as the data are loaded into memory. During system operation, the current frame depends on the contents of a segment register. Therefore, the frame provides a starting point (or base) that the linker can use in order to generate an accurate address when it performs the fix-up. If F=0, the FIXUPP record specifies the frame explicitly. If F=1, the FIXUPP record refers to the previous thread FIXUPP for the location of the frame.

The Frame field is a three-bit field whose meaning depends on the setting of the F field. If F=0 (an explicit frame), the Frame field contains a number from 0 to 5 that lists the method of identifying the frame. These methods of identifying frames (F0 through F5) are the same as those discussed earlier with the thread FIXUPP record.

If F=1 (reference to a previous thread), the Frame field identifies the number of a previously defined frame thread. For example, if the Frame field is 1, then the frame of this fix-up is frame thread number 1, as defined in the previous thread FIXUPP record.

The T field is a single bit that tells whether the target of this fix-up (the symbol to which the location refers) is specified explicitly (T=0) or in a previous thread FIXUPP (T=1).

The P field is a single bit that tells whether the target is specified in a primary way (P=0) or a secondary way (P=1). Primary ways correspond to methods T0 through T3 outlined earlier. They require both an index (to identify the segment, group, or external symbol) and an offset. Secondary ways correspond to methods T4 through T7 and do not require an offset (or the presence of the Target Displacement field in the FIXUPP record).

The Targt field is a two-bit field whose meaning depends on the value of the T field. If T=0 (explicit specification), Targt indicates the method of identifying the target. The possible values, 0 through 3, correspond to methods T0 through T3 if P = 0 and T4 through T7 if P = 1. These methods were described earlier in the discussion of thread FIXUPPs. If T=1 (reference to a previous thread), the Targt field identifies a previously defined target thread. In this case, the number is the same number of a target thread defined in the previous thread FIXUPP.

The Frame Datum field appears only when the frame is specified explicitly (F=1). It is an index into the list of SEGDEFs, GRPDEFs, or EXTDEFs (which list is used depends on the method identified in the Frame field).

The Target Datum field is similar to the Frame Datum field. It appears when the target is specified explicitly

TABLE 5: *Possible Method Values (D=0)*

VALUE	MEANING
0	The target is identified by a SEGDEF index and an offset. (T0)
1	The target is identified by a GRPDEF index and an offset. (T1)
2	The target is identified by an EXTDEF index and an offset. (T2)
3	The target is identified by a frame number and an offset. (T3)
4	The target is identified by a SEGDEF index only. The target starts at the beginning of the segment. (T4)
5	The target is identified by a GRPDEF index only. (T5)
6	The target is identified by an EXTDEF index only. (T6)
7	The target is identified by a frame number only. (T7)

The Method field lists the method the linker must use to identify the target or frame based on the value of the D field. IF D=0, Method can have any value shown here.

TABLE 6: *Possible Method Values (D=1)*

VALUE	MEANING
0	The frame is identified by a SEGDEF index. The frame is the highest-numbered frame that contains the segment. (F0)
1	The frame is identified by a GRPDEF index. (F1)
2	The frame is identified by an EXTDEF index. (F2)
3	The frame is identified by a frame number. (F3)
4	The frame is the highest-numbered frame that contains the segment of the location to be fixed up. The loader can determine this segment because the LEDATA record (which contains the location) indicates into which segment the data will be loaded. (F4)
5	The frame is the same as that of the target. (F5)

If the value of the D field is 1, indicating a frame thread, the Method field can have any one of the values shown in this list.

TABLE 7: *Possible Loc Values*

VALUE	MEANING
0	Lobyte (the low-order byte of a word).
1	Offset (the low-order word of a pointer).
2	Base (the high-order word of a pointer).
3	Pointer.
4	Hibyte (the high-order byte of a word).

The Loc field is a three-bit field that identifies the kind of location to be fixed up by the FIXUPP record. The possible values for this field are shown here.

(T=1) and is an index into the list of SEGDEFs, GRPDEFs, or EXTDEFs.

Finally, the Target Displacement field is a two-byte field that appears only when P=0 (the primary method of specifying a target). In this case, the Target Displacement is an offset from the start of the SEGDEF, GRPDEF, or EXTDEF (whose index appeared either in the Target Datum field or in an earlier thread FIXUPP).

The first record in the sample program serves as an example of how an explicit FIXUPP record works. This record starts at byte 1DFH. Its length (1AH 00H) is 26 decimal bytes. The next two bytes (CCH 08H) specify the Mode, Loc, and Data Rec Offset fields. The binary format of the hexadecimal numbers is as follows:

```
1  1  0  011  0000001000
```

The leftmost bit indicates that this actually is an explicit FIXUPP record (a 0 would have indicated a thread FIXUPP). The next bit, the Mode bit, indicates that the mode is segment-relative.

The third bit is always set to 0, and the three bits that follow make up the Loc field. The value of these bits (011 binary or 3 decimal) indicates that the location to be fixed up is a pointer.

The next 10 bits specify the location to be fixed up. This value (8 decimal) means that the location starts at the eighth byte from the start of the data portion of the previous data record (an LEDATA record starting at byte 19DH in figure 4). The data portion of that LEDATA record starts at byte 1A1H in figure 4, and the eighth byte is byte 1A8H (whose value is 04). This byte then becomes the first byte of a pointer the compiler has indicated for change.

The next byte in the FIXUPP record (56H) contains the values for the F, Frame, T, P, and Targt fields. To interpret these fields, this byte must be translated into binary, as follows:

```
0  101  0  1  10
```

The first bit is the F bit. Because it is set to 0, it indicates that this record specifies the frame explicitly, rather than by referring to a thread FIXUPP. Therefore, the next field (101) indicates the method of specifying the frame. Because 101 binary translates into 5 decimal, the method F5 is used. F5 says the frame is determined by the target. So for this fix-up, the information that specifies the target will also specify the frame.

The T bit is next. Because it is set to 0, it indicates that the target will also be specified explicitly and not by referring to a thread FIXUPP. The bit that follows is the P bit. Because it is 1, the target specification is made in the secondary way. That is, the index (in the Target Datum field) alone specifies the target. No displacement is needed, and therefore no Target Displacement field appears in this fix-up.

The last two bits in this byte (10) indicate the method of specifying the target. This value (2 decimal) means that method T6, an index into the list of external symbols, is used.

The next byte (0BH), the last one in this fix-up, is the Target Datum field. Notice that the Frame Datum field does not appear in this fix-up because the frame is specified implicitly by the target. The Target Displacement field does not appear either, because the target is specified in a secondary way. The remaining bytes in the record are additional complete fix-ups. Thus, this Target Datum field is an index that identifies both the target and the frame. Because the method of specifying the target was listed as T2, this is an index into the list of external symbols described in the EXTDEF record, and it identifies symbol 0BH (or 11 decimal). In the EXTDEF record (beginning at byte E2 in figure 4), the eleventh symbol listed is INIFQQ.

If the P bit had been set to 1, an additional displacement field would have identified the displacement from INIFQQ to which the fix-up should refer. Because P was set to 0, the location to be fixed up (identified earlier) depends solely on the address of the external symbol INIFQQ.

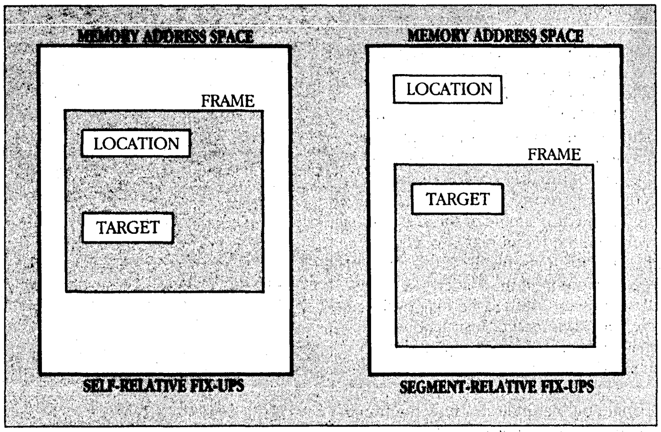
Thus, to summarize the first fix-up in this FIXUPP record, the location to be fixed up is a pointer that begins eight bytes into the previous data record and that points to the external symbol INIFQQ. When the location of INIFQQ becomes known (by linking the program to another module that declares INIFQQ as a public symbol), the linker will change the pointer to refer directly to INIFQQ. The linker knows exactly how to change the reference because it also knows the frame (the segment register value) that is in effect when the target is loaded into memory.

This FIXUPP record contains six more fix-ups. The one just outlined did not refer to any of the threads defined earlier, but some of the later ones do. To see some of the variations in fix-ups, just continue this process and translate some of the others shown in this listing.

BLKDEF and BLKEND records (offsets 332H and 346H). After some more LEDATA and FIXUPP records, a BLKDEF (block definition) and a BLKEND (block end) record appear. Together, these two records describe a program block. A program block can define such language constructs as procedures, loops, and multiline if-then-else statements, depending on the language and compiler implementation. A BLKDEF and a BLKEND record exist for every procedure and for every program block that has its own local variables.

The BLKDEF record identifies the group and segment containing the block. It also lists the name of the block, the block's offset from the start

FIGURE 6: *The FIXUPP Record*



FIXUPP records supply pieces of information the linker needs to fix up a location's reference. The relationship between these pieces of information is shown here.

of the segment, and the length of the block. In addition, if the BLKDEF record describes a procedure, the record provides information about the type of procedure. The format of a BLKDEF record is shown in figure 3-o.

The Group Index indexes the names in the LNAMES record. It identifies the group associated with the block. The Segment Index is an index into the list of SEGDEF records. It identifies the segment containing the block. If both these values are 0, a Frame Number field is present, indicating the number of the frame containing the block.

The Name field identifies the name of the block. As in all name fields, the first byte indicates the number of bytes in the rest of the name. The remaining bytes are ASCII values.

The Block Offset field is a word that indicates the block's offset from the start of the segment containing the block. The Block Length field is a word that lists the number of bytes in the block. The Procedure bit indicates whether the block is a procedure. If the bit is 1, the block is a procedure; however, if it is 0, the block is some different type (such as a DO loop).

The Long bit has meaning only if the Procedure bit is set to 1. When the Long bit is set to 1, it implies that the procedure's return address is a four-byte value (both CS and IP). When the bit is set to 0, the procedure has a two-byte return address (IP only).

The Return Address Offset field is present only if the block is a procedure. This field is a word that gives the location of the procedure's return address on the stack. The return address gets pushed onto the stack when the procedure is called. The Return Address Offset is interpreted as an offset from the BP register, which points to the return address on the stack.

As usual, the Type Index field identifies the number of the TYPDEF record that defines this block.

The BLKDEF record in figure 4 starts at byte 332H. The length (11H 00H) is 17 decimal bytes. The Group Index is 00H; thus, this block is not associated with a group. The Segment Index (01H) points back to the first SEGDEF defined, indicating this block is part of the segment WORDC.

The next field is the Name field. The first byte (05H) gives the number of bytes in the name. The next five (57H 4FH 52H 44H 43H) are ASCII codes for the block name (WORDC).

Next comes the Block Offset (01H 00H), indicating that the block starts one byte after the start of the segment.

TABLE 8: *Line Number/Line Number Offset Pairs*

SOURCE LINE NUMBER	OBJECT CODE OFFSET
11H (17 decimal)	**4EH**
12H (18 decimal)	**57H**
13H (19 decimal)	5DH
14H (20 decimal)	62H
16H (22 decimal)	**6FH**
17H (23 decimal)	**84H**
1BH (27 decimal)	A0H
1CH (28 decimal)	A7H
1EH (30 decimal)	**AEH**
1FH (31 decimal)	**B3H**
20H (32 decimal)	B7H
21H (33 decimal)	B7H
22H (34 decimal)	**B9H**
23H (35 decimal)	**EBH**

Each Line Number and Line Number Offset pair identifies the location of one line of source code. One such pair exists for each executable source line in the program.

TABLE 9: *Possible Attrib Values*

VALUE	MEANING
0	Non-main module with no starting address listed.
1	Non-main module with starting address listed.
2	Main module with no starting address listed.
3	Main module with starting address listed.

The Attrib field of the MODEND record is a two-bit field that specifies the attributes of the module. Possible values for this field are listed here.

The Block Length (EEH 00H) indicates that the block is 238 decimal bytes long. The next byte (60H) contains the settings for the Procedure and Long bits. In binary, this byte translates into 1100000. Therefore, this block is a procedure; appropriately, it does have a four-byte return address.

Because the block is a procedure, the next word (00H 00H) is the Return Address Offset. This value indicates that the procedure's return address is at BP+0. The byte that follows (00H) is the Type Index. The 0 value here indicates that no TYPDEF record is associated with this block.

A BLKEND record appears next, indicating the end of the block defined by the previous BLKDEF record. In the full Intel object module format, the combination of BLKDEF and BLKEND records provides information about the scope of variables. Between the BLKDEF and BLKEND records appear debugging symbol records that describe each symbol defined in the block. However, the Microsoft format does not currently support these debugging records; BLKEND records therefore serve no useful purpose.

The format of a BLKEND record is shown in figure 3-p and contains no special fields. The BLKEND record in figure 4 starts at byte 346H.

LINNUM record (offset 34AH). After the BLKEND record comes a line numbers record that lists the address of each executable line of source code. Although not necessary for program execution, it provides information debuggers can use to associate the source code with the translated object code.

The format of a LINNUM record is shown in figure 3-q. In this record, the Group Index field is an index into the LNAMES record and indicates the group containing this code. Likewise, the Segment Index is an index into the list of SEGDEF records, identifying the segment containing the code.

Each Line Number and Line Number Offset pair identifies the location of one line of source code. The Line Number field is a word that lists the source line number. The Line Number Offset field is a word that lists the address of that line, relative to the start of the segment. A Line Number/Line Number Offset pair exists for each executable source line in the program.

The LINNUM record of the sample program starts at byte 34AH of figure 4. Its length (3BH 00H) is 59 decimal bytes. Its Group Index is 00H (no group), and its Segment Index is 01H (the first SEGDEF, WORDC). Next come the Line Number/Line Number Offset pairs. These pairs are listed in table 8.

To see those lines included in this record, compile the source code in listing 1 and look at the file WORDC.LST.

MODEND record (offset 388H). The last record in the file is the module end record. It indicates the end of the module that was begun by THEADR, whether that module is a main module or a subprogram, and sometimes the module's starting address. Figure 3-r shows the record's format.

The Attrib field of the MODEND record is a two-bit field that specifies the attributes of the module. Possible values for this field are listed in table 9.

The rest of the fields (F, Frame, T, P, Targt, Frame Datum, Target Datum, and Target Displacement) specify the starting address. These fields appear only if the Attrib field indicates that the starting address is listed. The fields are interpreted in the same way as the equivalent fields in the FIXUPP record.

In figure 4, the MODEND record starts at byte 388H. The length of the record (02H 00H) is two bytes. The Attrib field is 0, which indicates a non-main module with no starting address listed. The other fields are not present. If this sample program had included more than one module, another THEADR record would have appeared to indicate the start of the next module.

THE EXPANDED STANDARD

Even though Microsoft defines 15 different record types, these are just a subset of the record types available in the full Intel set (see table 2). What do these extra records provide, and why did Microsoft choose to omit them?

Some of the records (such as PEDATA, PIDATA, REDATA, RIDATA, RHEADR, LHEADR) indicate kinds of relocatable or absolute data that Microsoft deals with in other ways. For example, Intel's relocatable object modules (REDATA and RIDATA) correspond to Microsoft's .EXE files. Intel's absolute object modules (PEDATA and PIDATA) correspond to Microsoft's .COM files. The correspondence is not exact, but .EXE and .COM files provide enough features that Microsoft does not need to support those additional records.

Other object records, such as the full Intel implementation of TYPDEF, DEBSYM, or LOCSYM, are debugging records with information about the source program. Such information allows symbolic and high-level language debuggers to provide additional services to programmers, such as displaying local variables or producing formatted dumps of structures.

Two probable reasons Microsoft decided not to include these debugging records are code size and compiler development time. The object files generated by the Intel compilers contain more information, so they are larger. In the early days of the PC, when DOS supported only single-sided (160KB) floppy disks, it was more important to ensure that a program would fit on a disk than it was to provide elaborate debugging information. Therefore, all unnecessary records were eliminated. Although larger-capacity disks are now supported by the PC, most compilers and linkers have not yet been modified to add the extra records.

The other reason for the lack of these extra records might be the time necessary to write a compiler. Requiring a compiler to generate extra records means additional effort by the compiler writer. Further, because IBM DEBUG, the first debugger for the PC (and the only one for a long time) does not use this information, adding records probably seemed like wasted effort. Omitting the extra records got the compilers out to the public sooner with no noticeable loss of features.

Today's market, however, demands debugging tools that are more advanced than IBM DEBUG. High-level language debuggers, which allow programmers to debug while viewing source code (not an assembly language version of it), are possible, but they require that object modules contain more information than current PC compilers now generate. To lay the groundwork for these advanced debugging tools, Microsoft and other companies soon may add debugging records to the object modules generated by their compilers.

The format of the object module is the key to compatibility between different compilers, assemblers, linkers, and librarians. Products from many different companies use the same format (the one recognized by the LINK program). This standard format allows programmers to combine code generated by many different products.

Besides linking object modules, programmers can also combine them into libraries so that the individual modules can be pulled out selectively by the linker. These library files contain additional records that provide this selective linking capability.

REFERENCES

Intel Corporation. *8086 Relocatable Object Module Formats. An Intel Technical Specification.* (Santa Clara, CA: Intel Corp., 1981).

Microsoft Corporation. *MS-DOS Programmer's Reference Manual.* (Bellevue, WA: Microsoft Corp., 1984).

Steven Armbrust is a freelance technical writer. Ted Forgeron is a microcomputer software consultant. They work primarily in the "Silicon Forest" near Portland, Oregon.

LISTING 1: WORDC.PAS

```
{ WORDC -- Counts the number of words in a file }

PROGRAM wordc(in_file,output) ;

  CONST
    tab = 9 ;
    lf = 10 ;
    cr = 13 ;

  VAR
    in_file,out_file : text ;
    count : integer ;
    input_char : char ;
    inword : boolean ;

  BEGIN {wordc}
    reset(in_file) ;
    count := 0 ;
    inword := false ;
    WHILE NOT eof(in_file) DO
      BEGIN
        read(in_file,input_char) ;
        IF (input_char=' ') OR
           (input_char=chr(tab)) OR
           (input_char=chr(cr)) OR
           (input_char=chr(lf)) THEN
          inword := false
        ELSE IF ( NOT inword) THEN
          BEGIN
            inword := true ;
            count := count + 1 ;
          END ;
      END ; {while}
    writeln('word count = ',count) ;
  END. {wordc}
```

Microcomputer Software Development Tools

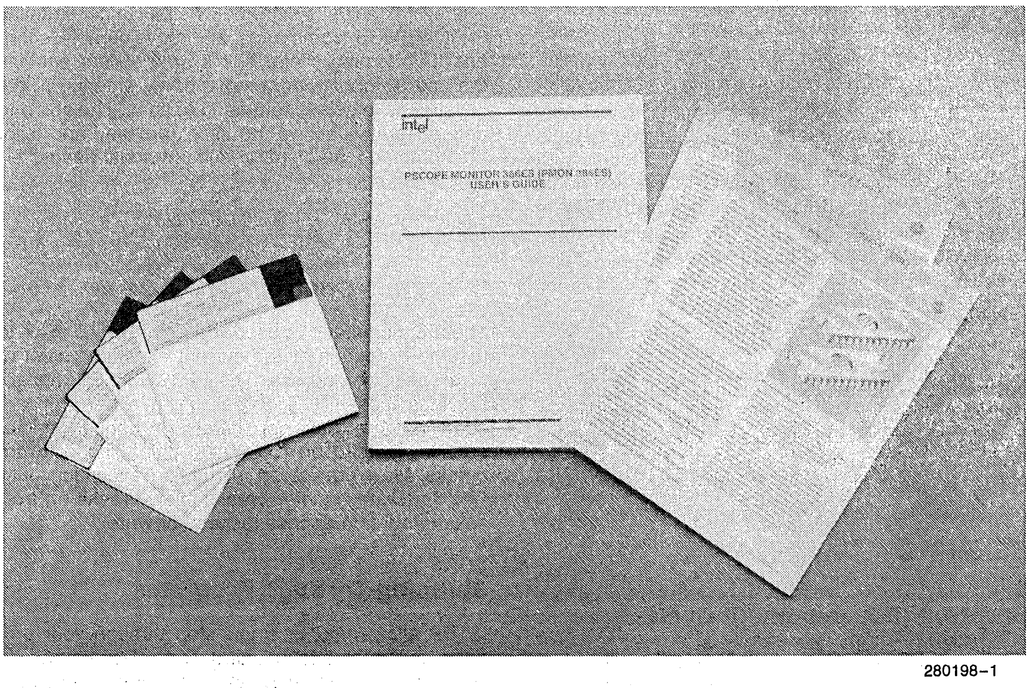
2

PSCOPE MONITOR 386ES (P-MON386ES)

- **Program Execution Control, Including Breakpoints and Single-Step Execution through Assembly Level Instructions in Both Real and Protected Mode**
- **Examine and Modify Memory, I/O Ports, and Processor Registers**
- **Examine and Modify Descriptor Tables and the Task State Segment**
- **Download 8086, 80286, and 80386 code in Intel Object Module Format**
- **Disassemble Memory in 386 Instruction Mnemonic Form**
- **Host Software Executable on the Intel System 286/310 with Intel XENIX* Release 3.0, Update 3**
- **User-Friendly Human-Interface Provides Command Line Editing, On-Line Syntax Guide, On-Line Syntax Builder, and a Command History**

The Intel PSCOPE Monitor 386ES (P-MON386ES) is a debug monitor for 80386-based systems, and is designed to provide software development aid for systems programmers. It can access and control all of the 80386 visible user-hardware resources without any assistance from an operationg system. With the help of this monitor, a user can download a program into the target prototype memory, set hardware and software breakpoints, examine/modify memory and processor registers, and control program execution. This monitor is supplied with a serial driver for the 8251 and the 8274, but it can be configured to run on any 80386-based target board with a user-supplied communication driver.

The P-MON386ES package includes the diskettes that contain the host software to be loaded into the Intel System 286/310, the diskettes that contain the target software to be loaded into the 80386-based target board, and the *PSCOPE Monitor 386ES User's Guide,* order number 166184-002.



280198-1

*XENIX is a trademark of Microsoft Corporation.

October 1986
Order Number: 280198-001

FUNCTIONAL DESCRIPTION

Overview

P-MON386ES provides early design aid and debug support for Intel customers who are designing software to run on the 80386 high performance microprocessor with integrated memory management. P-MON386ES is hosted on the Intel System 286/310 with XENIX release 3.0, with Update 3, and allows the user to perform the following tasks:

- Download Intel 8086, 80286, and 80386 object module formats (with no symbolics)
- Examine/modify memory, I/O ports, processor registers, descriptor tables, and the task state segment
- Convert addresses from virtual to linear, linear to physical, and virtual to physical
- Evaluate expressions
- Control execution both in real and protected mode
- Set software breakpoints on execution addresses
- Set hardware breakpoints on execution and data addresses
- Disassemble memory

Formatted Displays

The P-MON386ES allows the user to view all of the 80386 visible hardware resources and pre-defined data structures in easy-to-read formats. These displays include contents of the global descriptor table (GDT), the local descriptor table (LDT), the interrupt descriptor tables (IDTs), the task state segments (TSSs), the extended flags, register (EFLAGS), the segment registers (SR), and the set of the most commonly accessed 80386 registers (REGS).

Command Execution

Figure 1 illustrates a typical software development environment with the 80386-based target board connected to the Intel System 286/310 via a serial link. P-MON386ES commands are entered interactively from the terminal attached to the Intel System 286/310.

BENEFITS

Shortened Development Cycle

With P-MON386ES, you can use control constructs, which provide repetitive or conditional execution of P-MON386ES commands, and you can make code patches by directly writing to memory. These features help to shorten the development cycle by easily isolating software bugs and by quickly testing program changes.

Improved Debugging Productivity

With the P-MON386ES monitor you can display and modify program variables. In addition, it allows you to define, display, modify, and remove debugger objects (such as break registers and literallys).

More Reliable Software

The control constructs can be used to repeatedly generate test values, execute the program with input values, and record the results. Running more comprehensive tests yields more reliable software.

Easy to Learn and Use

An extensive command language, which is similiar to block-structured languages such as PL/M, Pascal, and C, is very easy to use in an interactive debug session. The operators and control constructs are similiar to those in the C programming language. I^{2}ICE™ system-like syntax is also provided.

The syntax guide is extremely helpful in constructing P-MON386ES commands and considerably shortens the learning cycle. The syntax builder enables the user to construct commands by entering single keystrokes to select command options. The ability to define literallys allows the user to extend and tailor the command language to suit individual needs.

FEATURES

Execution Control

The GO command is used to begin execution or to resume execution at the current execution point or at a specified address. The P-MON386ES monitor provides the ability to break using either software code patch breaks or hardware debug register breaks provided on the 80386 chip. The software breaks allow breaks on instruction execution only. The hardware register breaks allow breaks on execution and data access (access or write).

Stepping Through Programs

ISTEP allows you to single-step through your program by machine-level instructions. Tracing can be simulated by entering the command sequence ISTEP; ASM CS:EIP.

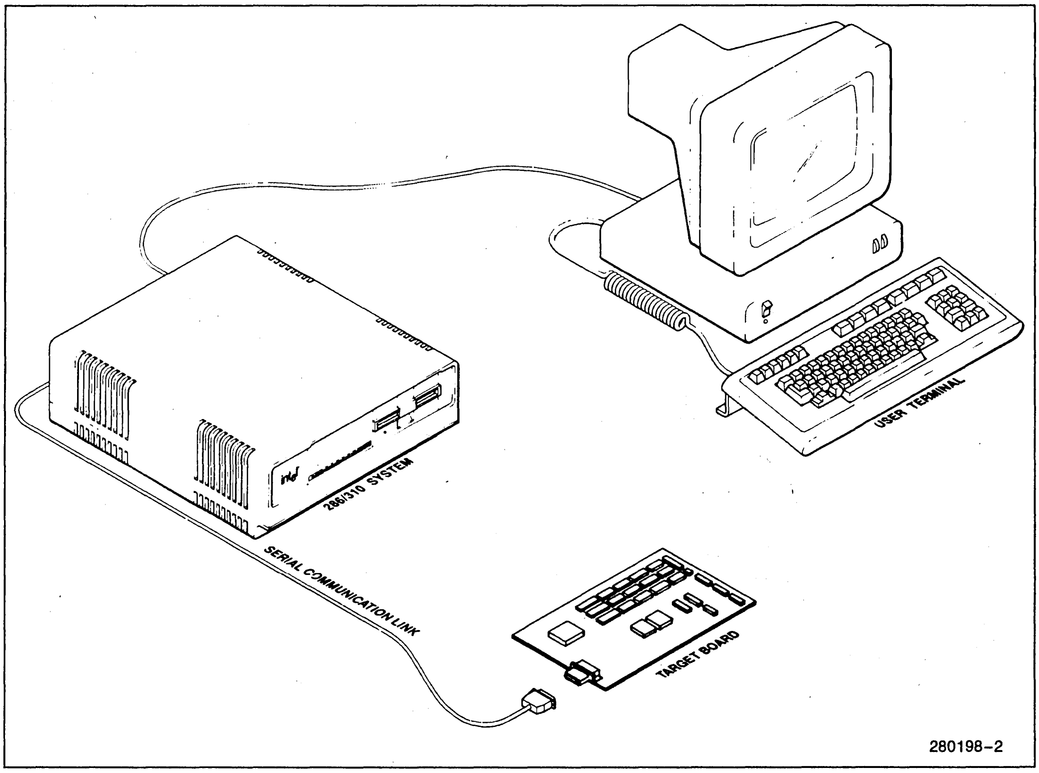


Figure 1. Software Development System

Debugger Command Language

ASM/USE—For disassembling code

DO/END—For defining command blocks

DT/GDT/LDT/IDT—For allowing access to descriptor tables

ECHO—For console output

FOR, IF/THEN/ELSE, WHILE and UNTIL—For conditional execution of commands or blocks

GO/SWBREAK/SWREMOVE/ISTEP—For controlling program execution

INCLUDE/SAVE/LOG/NOLOG—For saving/restoring commands and definitions to and from disk

ORD1/ORD2/ORD4/INT1/INT2/INT4—For accessing memory

PHYSICAL/LINEAR—For converting addresses to physical or linear addresses

PORT/DPORT/WPORT—For accessing I/O ports

REGS/80386 registers—For accessing all of the 80386 registers

REPEAT/COUNT—For repetition of commands or blocks

SWITCH—For branching execution to one of several case statements

TSS—For accessing the task state segment

SPECIFICATIONS

Host Development System Environment

The host development system for the P-MON386ES requires the following minimum configuration, which can support up to two P-MON users:

- Intel 286/310 System
- XENIX Release 3.0, Update 3

- 2M bytes of main memory
- 40M-byte hard disk
- Intelligent four-channel communication controller

Target System Environment

The P-MON386ES is designed to be used in any 80386-based target system that includes the following:

- At least 80K bytes of EPROM space
- At least 16K bytes of RAM space
- A serial communications interface

The starting locations for the software loaded into the EPROMs and RAM space is user-specified.

ORDERING INFORMATION

Order Code	Description
X286PMON386ES	The P-MON386ES monitor package includes host and target software diskettes and a user's guide. The host software is shipped on 5¼″ 360KB TAR format diskettes and the target firmware is shipped on 5¼″ 360KB DOS diskettes, 5¼″ iNDX diskettes.

The purchase of the P-MON386ES package includes a 90-day support service, which also includes the Software Problem Report Service.

Registered customers will be automatically upgraded with the pre-production version of this product on release.

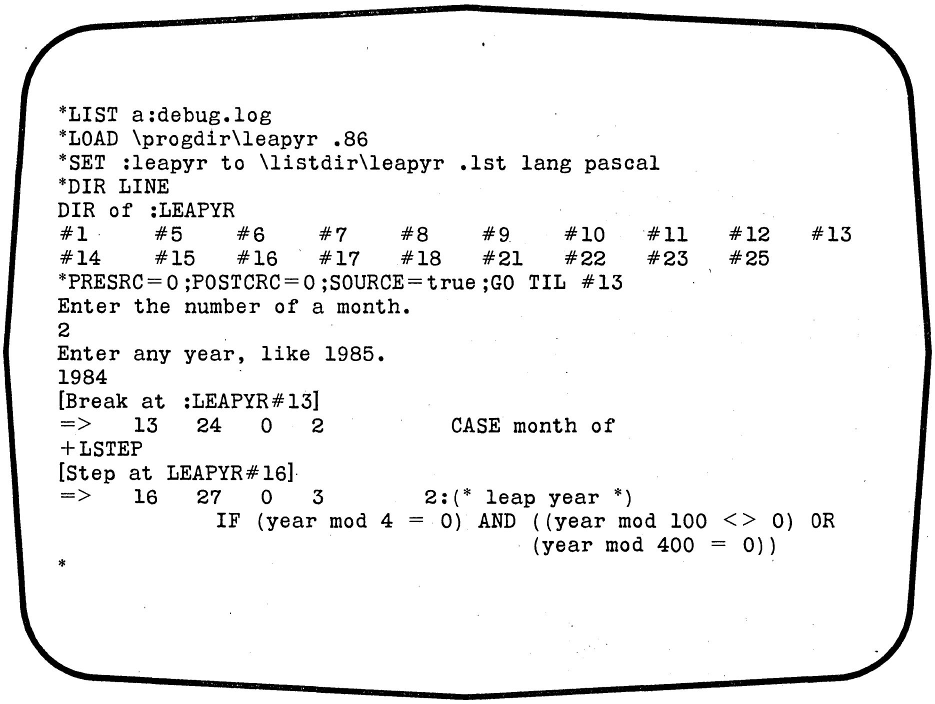
As with all Intel software, purchase of any of these options requires the execution of a standard Intel Master Software license. The specific rights granted to users depends on the specific option and the license signed.

intel®

PSCOPE-86 FOR DOS HIGH-LEVEL APPLICATION PROGRAM DEBUGGER

- **Debugs PL/M-86, Pascal-86, iC-86, FORTRAN-86, and ASM86 Programs**
- **Displays Program Text on the Screen During Debugging:**
  - **— Uses the Listing File to Display Program Text**
  - **— Displays Source Code on Program Step, at Execution Break Points, or on User Request**
- **Disassembles Memory and Provides an Interactive Assembler**
- **Permits Creation of Program Patches Using High-level Language Constructs**
- **Supports Access to DOS Operating System Commands**
- **Offers Symbolic Debugging Capabilities:**
  - **— Supports Access to Memory by Program Defined Variable and Program Names**
  - **— Maintains Type Information About Variables**
  - **— Allows Definition of User-defined Debugging Variables and Procedures**
- **Single-steps Through Assembly Language Instructions, High-level Language Statements, or Procedures**
- **Sets Break Points and Traces Program Execution**
- **Runs Under the PC-DOS Version 3.0 or Greater**

PSCOPE-86 for DOS is an interactive, symbolic debugger for high-level language programs written in iC-86, PL/M-86, Pascal-86, and FORTRAN-86, and for assembly language programs written in ASM86. PSCOPE-86 for DOS runs under the PC-DOS operating system, version 3.0 or greater.



```
*LIST a:debug.log
*LOAD \progdir\leapyr .86
*SET :leapyr to \listdir\leapyr .lst lang pascal
*DIR LINE
DIR of :LEAPYR
#1      #5     #6     #7     #8     #9     #10    #11    #12    #13
#14     #15    #16    #17    #18    #21    #22    #23    #25
*PRESRC=0;POSTCRC=0;SOURCE=true;GO TIL #13
Enter the number of a month.
2
Enter any year, like 1985.
1984
[Break at :LEAPYR#13]
=>  13   24   0   2          CASE month of
+LSTEP
[Step at LEAPYR#16]
=>   16   27   0   3      2:(* leap year *)
             IF (year mod 4 = 0) AND ((year mod 100 <> 0) OR
                                      (year mod 400 = 0))
*
```

280194-1

October 1986
Order Number: 280194-002

MAJOR FEATURES

With PSCOPE-86 for DOS, a user can load an application program, set break points at symbolic or numeric addresses, trace program execution, and view source code text. Program bugs can be patched using high-level PSCOPE commands or assembly code. The corrections can be tested without leaving the PSCOPE software.

Other debugging aids include the ability to single-step a program through assembly language instructions, high-level-language statements, or procedures, to display and modify program variables, to inspect files, and to personalize the debugging environment.

The following sections describe some of the major features of PSCOPE-86 for DOS.

Source Display

With the DOS version of PSCOPE-86, a user can correlate a module under debug to a source code file. Then, when break points are encountered, source text is displayed along with the break message and line number of the break point. The number of source lines displayed before and after a break point can also be defined by the user.

View all or part of the listing file on command. The following example uses the PSCOPE command to list the current module. The asterisk (*) is the PSCOPE prompt, the command follows, and after pressing <Enter>, PSCOPE responds with a list file.

```
*SHOWSRC #1 LENGTH 28
 1   1   0   0  program leapyr (input,output);
                (* Input month and year, receive number of days *)
 2   5   0   0  var year    :integer;
 3   6   0   0      month   :integer;
 4   7   0   0      nrdays  :integer;

 5   9   0   0  begin

 5  11   0   1     month := 0;
 6  12   0   1      year := 0;
 7  13   0   1    nrdays := 0;

 8  15   0   1  writeln('Enter the number of a month.');
 9  16   0   1  readln(month);
10  17   0   1    while month <> 999 do

11  19   0   1    begin

11  21   0   2  writeln('Enter any year, like 1985.');
12  22   0   2  readln(year);

13  24   0   2      CASE month of
14  25   0   3                  4,6,9,11:nrdays := 30;
15  26   0   3            1,3,5,7,8,10,12:nrdays := 31;
16  27   0   3                         2: (* leap year *)
                    IF (year mod 4 = 0) AND ((year mod 100 <> 0) OR
                                             (year mod 400 = 0))
                                   THEN nrdays := 29
17  31   0   3                     ELSE nrdays := 28;

19  33   0   3      end;

21  35   0   2    writeln('Number of days in the month is',nrdays);

22  37   0   2    writeln('Enter the number of a month.');
23  38   0   2    readln(month)
                 end;
25  40   0   1  end.
```

Single-Stepping

PSCOPE has two commands to single-step through high level instructions and display source code. The commands differ in how they handle program calls. The following example illustrates the LSTEP command.

```
*LSTEP
[Step at :LEAPYR#17]
=>  17  31  0  3                     ELSE nrdays := 28;
    19  33  0  3              end;
*LSTEP
[Step at :LEAPYR#21]
=>  21  35  0  2        writeln('The number of days in the month is',nrdays);
```

PSCOPE can single-step through code at assembly level and display assembly mnemonics as in the following example which uses the ISTEP command.

```
*ISTEP
:LEAPYR
512A:00FEH      C70600000000       MOV WORD PTR 0000H,0
```

Symbolic Debugging

With symbolic debugging, a user can examine or modify a memory location by using its symbolic reference. A symbolic reference is a procedure name, variable name, line number, or program label that corresponds to a location in the user program's memory space. For example, to display the value of the program variables, users need only execute the program until the variable is active and type that variable's name.

```
*LSTEP
[Step at :LEAPYR#22]
=>  22  37  0  2           writeln('Enter the number of a month.');
*month
+2
*year
+1900
*nrdays
+28
```

Define the Debug Environment

With the PSCOPE high-level program debugger, a user can define the debugging environment within PSCOPE software. You can define break points and trace points. With PSCOPE, you can write macros that set the debug environment when PSCOPE is invoked, or these macros can be included at any time during the debugging session. Shorten commands with literal definitions, try program bug fixes with patches and procedures, or write procedures to control program execution. All debug variables and procedures can be saved in files and reused.

BREAK REGISTERS AND TRACE REGISTERS

Breaks occur at addresses in the program under execution. The user can enter physical addresses or symbolic addresses to halt program execution. With PSCOPE, you can easily break at executable statement addresses by using line numbers. Simply use the PSCOPE directory command with the line option (DIR LINE) to get a directory of line numbers. Then define a break register or a trace register to stop at these addresses.

A break register (BRKREG) stops program execution and returns a PSCOPE prompt (*). A trace register (TRCREG) displays a message and continues program execution. Following are examples of how to define a break register and a trace register.

```
*DEFINE BRKREG stop = #22
*DEFINE TRCREG stop2 = #17
```

DEBUGGING PROCEDURES

Debugging procedures are groups of PSCOPE commands that have been labeled. Writing procedures with PSCOPE commands is much the same as writing high-level language procedures. A procedure can be used for any definable function during a debugging session, and it can be used with a program under execution.

In the LEAPYR program, the while loop continues until 999 is entered for a month number. The following example of a PSCOPE procedure (PROC) that querys the user about halting execution. If the answer is yes ('Y' or 'y'), the procedure sets *month* to 999.

```
*DEFINE PROC query = DO
.*WRITE USING ('Do you want to quit? Enter Y for yes.')
.*DEFINE CHAR ccc = CI
.*WRITE ccc
.*IF ccc =='Y' or 'y' then
..*month == 999
..*RETURN = true
..*else RETURN = false
.*endif
.*END
*
```

To call this procedure while the program is executing, define a break register and use it with the GO command as follows:

```
*DEFINE BRKREG stop = #22 CALL query
*GO USING stop
```

PSCOPE PATCHES

A PSCOPE patch is used to temporarily correct run-time errors in the program under debug. A patch can be an additional line (or lines) in a program, or can be used to replace lines in a program. PSCOPE enables both high-level patches (the PATCH command) and assembly-level patches (the ASM command).

High-Level Patch

In the LEAPYR program, the way to exit the program is to enter 999 for the month. However, nothing instructs the user to do this. With a high-level patch, it is simple to add a line of code to the program. Following is an example.

```
*DEFINE PATCH #22 = WRITE 'To exit the program, enter 999.'
```

When the program is executed, the patch is used automatically. There is no need for a break register. Program execution stops at line number 22, the patch message is displayed, and program execution continues at line number 22. It is also possible to replace lines by using the TIL option in a high-level patch. Then program execution continues from the line number, or address, defined after the TIL. To simply eliminate lines of code, set the line to NOP as follows:

```
*DEFINE PATCH #18 = NOP
```

Assembly-Level Patch

Assume there is a typo in the LEAPYR program. Instead of the else condition setting nrdays to 28, it sets nrdays to 29, making every year leap year. Use the ASM command first to display assembly code as in the following example.

```
*ASM #17 LENGTH 4
:LEAPYR
521A:01E0H    C70600001D00       MOV  WORD PTR 0000H,001DH   ;+29T
521A:01E6H    EB06               JMP   $+0008H             ; A=01EEH
=>  17  31  0 3                     ELSE nrdays := 29;
521A:01E8H     C70600001D00      MOV  WORD PTR 0000H,001DH   ;+29T
521A:01EEH EB00 JMP $+0002H            ; A=01F0H
```

Notice that source code can be displayed to assist you in finding the ELSE statement. However, source display can be eliminated simply by setting the variable SOURCE to false. After finding the address for the correct line of code, use the ASM command to change the second 29 to 28. Notice in the following example, 'word' is sufficient for the assembly mnemonic. The 'ptr' mnemonic is unnecessary.

```
*ASM 521A:01E8H = 'mov word 0000H,001Ch'
521A:01E8H    C70600001C00      MOV  WORD 0000H,001DH
```

LITERALLY DEFINITIONS

LITERALLY definitions are shortened names for previously defined character strings. LITERALLY definitions save keystrokes or improve clarity. For example, the following LITERALLY definition replaces the command DEFINE with the abbreviation DEF.

```
*DEFINE LITERALLY def = 'DEFINE'
```

Save and Restore the Debug Environment

All debug variables and procedures can be saved in a file for future debug sessions. To save everything in a file, use the PUT command as follows:

```
*PUT a:debug.mac DEBUG
```

The saved file can be used as a macro and invoked automatically with PSCOPE by using the following invocation command to start PSCOPE.

```
C:>PSCOPE MACRO(a:debug.mac)
```

After PSCOPE is loaded, a list of all the commands in the macro will print to the screen and will be included in the debug environment. It is also possible to include a macro after PSCOPE is loaded. The following example uses the NOLIST option to prevent the commands from writing to the screen.

```
*INCLUDE a:debug.mac NOLIST
```

The Internal Editor

PSCOPE has an internal editor that is a version of Intel's Aedit. Use this editor to correct source code as program fixes are confirmed with PSCOPE. The editor can also be used to create macros, procedures, or correct command lines.

Escape to DOS

PSCOPE has an escape function to enable access to the DOS operating system commands. This is very useful to verify a file location or print a file. Any DOS operating system command is accepted after entering the 'bang', explanation point, (!). The following is an example of the ESCAPE command.

```
* !print a:debug.mac
```

The DOS print message will appear on the screen, and then the PSCOPE prompt. Once the printing is complete, you are again in PSCOPE withoutaltering the debug environment.

The PSCOPE Command Language

The syntax of PSCOPE commands resembles that of a high-level language. The PSCOPE command language is versatile and powerful while remaining easy to learn and use because commands are often self explanatory like GO. GO starts execution of the user program.

The PSCOPE command language can be divided into functional categories.

- Emulation commands instruct PSCOPE to execute the user program. They consist of GO and the three stepping commands, ISTEP, LSTEP, and PSTEP.
- Debugging environment commands define PATCHes, debugging PROCedures, debugging *variables*, LITERALLYs, break registers (BRKREG), and trace registers (TRCREG) using the DEFINE command. A user can also delete these definitions with the REMOVE command.
- Block commands consist of DO-END, COUNT-END, REPEAT-END, and IF-THEN-ELSE constructs. They can be used alone or within debugging procedures and patches.
- String functions concatenate strings (CONCAT), return the string length (STRLEN), return a substring (SUBSTR), and accept console input (CI).
- Utility commands are general-purpose commands for use in a debugging environment. They consist of the following:

!	accesses the DOS operating system commands.
$	is a pseudo-variable that represents the current execution point.
ACTIVE	is a function that determines whether a specified dynamic variable is currently defined on the stack.
ASM	assembles or disassembles memory.
BASE	sets or displays the current radix.
CALLSTACK	displays the dynamic calling sequence stored on the stack.
DIR	displays all objects of a specified type.
EDIT	invokes the internal, menu-driven text editor.
EVAL	returns the value of a symbol in binary, decimal, hexadecimal, and ASCII.
EXIT	returns control to the host operating system.
HELP	provides on-line help for selected topics and selected error messages.
NAMESCOPE	is pseudo-variable that represents the current scope of a variable. It gives access to variables without requiring a fully qualified symbolic reference.
OFFSET$OF	is a function that returns the offset of a specified address (virtual or symbolic).
SELECTOR$OF	is a function that returns the selector of a specified address (virtual or symbolic).
WRITE	writes variables and strings to the console's screen.

- File handling commands access disk files. The user can load program files to be debugged (LOAD), save patches, debugging procedures, debugging variables, LITERALLYs, and debugging registers in a disk file (PUT and APPEND), read-in these definitions during later debugging sessions (INCLUDE), and record a debugging session in a disk file for later analysis (LIST and NOLIST).
- Register access commands provide access to the 8086/8088 registers and flags.

  The REGS command displays the 8086/8088 registers and flags. Users can also inspect or change an individual register by specifying its mnemonic. For example, CS represents the code segment register.

  The FLAG pseudo-variable represents the 8086/8088 flag word. The user can also inspect or change each flag separately as a Boolean variable. (For example, TFL represents the trap flag).

  PSCOPE provides register access for programs that perform real arithmetic. There is a built-in 8087 math coprocessor emulator, or there is a CH8087 option with the LOAD command to tell PSCOPE to access the hardware (8087 math coprocessor chip) registers. Access or change the 8087 registers by name.
- Source display commands are used to view a specified number of lines of source text at break points or on demand. LPATH or SET directs PSCOPE to the source text file. SOURCE is the pseudo-variable used to determine if source text will be displayed at break points. With PRESRC and POSTSRC, the user can determine how many lines of source code will be displayed before and after the line at the break point. The SHOWSRC command enables the display of source code outside of program execution.

SPECIFICATIONS

Memory Requirements

PSCOPE-86 for DOS requires approximately 300KB of memory for PSCOPE software and buffers.

DOS Version

PSCOPE is designed to run on the DOS operating system version 3.0 or greater.

Language Support

iC-86

PL/M-86

FORTRAN-86

ASM86

PASCAL-86

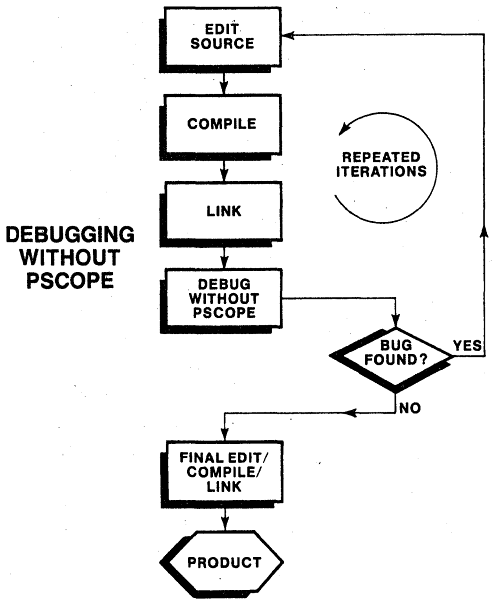
ORDERING INFORMATION

Order Code	Description
D 86 PSC 86	High-Level Software Debugger

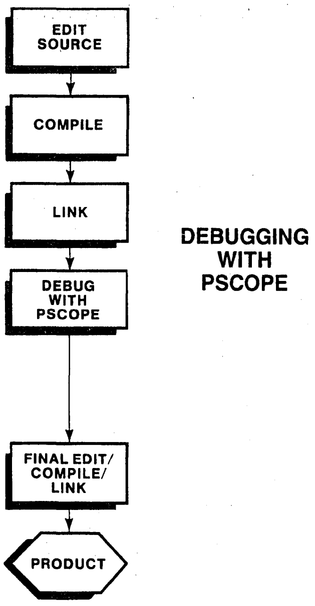
PSCOPE HIGH-LEVEL PROGRAM DEBUGGER FOR iRMX™, XENIX*, SERIES III AND SERIES IV

- **Provides Source Level Debugging Capabilities for High-Level Languages and Assembly-Level Languages**
- **Permits Creation of High-Level Program Patches using PSCOPE High-Level Language Constructs**
- **Sets Breakpoints and Traces Program Execution**
- **On-Line Help Facilities**
- **Code Disassembly/Assembly and Assembly-Level Patching**
- **Symbolic Debugging Capabilities**
  - **—Maintains Type Information about Variables**
  - **—Supports Symbolic Access to Dynamic Local Variables**
  - **—Maintains a Virtual Symbol Table for Program Variables**
  - **—Allows Definition of User-Defined Debugging Variables and Procedures**
  - **—Accesses Memory Locations and Program Variables using Program-Defined Names**
- **Single Stepping**

PSCOPE is an interactive, symbolic debugger for high-level-language programs written in PL/M, Pascal, and FORTRAN. The iRMX™ PSCOPE and XENIX™ PSCOPE products support debug of programs written in assembly language. XENIX PSCOPE provides additional C language support and can use XENIX operating system commands without leaving or altering the debug environment.



280266-1



280266-2

*XENIX is a trademark of Microsoft Corporation.

October 1986
Order Number: 280266-002

PSCOPE OVERVIEW

With PSCOPE, a user can load an application program into host system memory, set breakpoints at symbolic or numeric addresses, trace program execution, and create patches. Other debugging aids include the ability to single-step a program through high-level-language statements or procedures, to display and modify program variables, to inspect files, and to personalize the debugging environment.

MAJOR FEATURES

The following sections describe the major features of the PSCOPE high-level debugger.

Symbolic Debugging

With symbolic debugging, a user can examine or modify a memory location by using its symbolic reference. A symbolic reference is a procedure name, variable name, line number, or program label corresponding to a location in the user program's memory space. For example, to display the value of the program variable *linesend*, users need only GO TIL the variable is active, and type the variable's name. Note that * is the PSCOPE prompt.

```
*linesend
50
```

Notice that PSCOPE returns the variable value without the user having to indicate the variable's type. The capability to recognize a variable's type and scope is a special feature of PSCOPE's support of symbolics. Few other debuggers offer this feature.

Consider another example. Suppose the user's program has an array of employee records called *emprec* that includes salary and other employee information. Using PL/M, the user might declare it as follows:

```
DECLARE emprec (100) STRUCTURE
                    (name (20) BYTE,
                    ss (10) BYTE,
                    number INTEGER,
                    salary REAL);
```

With PSCOPE, to determine the salary of the nth employee, the user need only type:

```
emprec[n].salary
```

PSCOPE would then respond:

```
2.200E + 03
```

Patch

A patch is a set of PSCOPE high-level commands that augments or replaces a section of the user's program. With patching, the user can modify a program's algorithm and verify the effect of modifications without having to edit source, recompile and relink.

For example, the following patch indicates that if the value of the variable *x* is 0, then the current execution point ($ is a PSCOPE pseudo-variable for the current execution point) will be line #39. If *x* is not 0, the value of *y* will be set to the value of *linesend* minus the value of *x*. The execution point will not change.

```
*DEFINE PATCH #37 TIL #39 = DO
.*IF x = 0 THEN $ = #39
.*ELSE
.*y = linelength - x
.*ENDIF
*END
*
```

A patch can also be used to bypass statements. The following command causes lines #13 through #15 to be skipped, resuming execution at line #16.

```
*DEFINE PATCH #13 TIL #16 = nop
```

Breakpoints

Breakpoints suspend program execution at specified locations. The user can then enter PSCOPE commands, construct patches, examine or change program variables and registers. PSCOPE for the XENIX and iRMX operating systems also allow memory disassembly. Execution can be resumed from the breakpoint or from any other point.

A breakpoint specification is the address where program execution stops. The address can be specified as a symbolic address, segment/offset pair, or as a high-level-language statement number.

For example:

```
*GO TIL: count_lines
[Break at count_lines]
```

Debug Procedures

A debug procedure is a group of PSCOPE commands that are invoked by a name. Debug procedures can be saved and then recalled for use in later debug sessions.

Following is the definition of a debug procedure called *sum*. It consists of a DO command block that returns the sum of all the parameters passed to it. It contains two local variables, *n* and *i*.

```
*DEFINE PROC sum = DO
.*DEFINE LONGINT n = 0
.*DEFINE INTEGER i = 0
.*COUNT % np
..*n = n + %(i)
..*i = i + 1
..*ENDCOUNT
.*RETURN n
.*END
*
```

To execute the debug procedure, invoke it by name as follows:

```
*sum(2,3,5)   /*Executing the debug procedure*/
```

The benefits of debug procedures include simplification of command invocation for groups of commands and automation of the software verification process. A procedure can be defined to iteratively generate test values, execute the program with new input values, and record results. Thus, debug procedures can be used to develop and run comprehensive "batched" tests.

Break Registers

A break register is a named set of one or more breakpoint specifications. After defining the contents of break registers, the user can execute a program using the specifications in one or more of these break registers. For example, here is the definition of a break register called *break1*:

```
*DEFINE BRKREG break1 = error_check
```

To execute and break just before procedure error_check, specify use of break register *break1* in the GO command.

```
*GO USING break1
```

Break registers are useful for storing sets of specifications to be recalled in later debug sessions. They also enable users to call a previously defined debugging procedure when a specification in a break register is met. This gives the user a powerful tool for creating conditional constructs to observe program behavior. For example, assume a user wants to break at three breakpoints, *term, value,* and at line #68. Before breaking at the procedures *term*, and *value*, the user wants to execute a debug procedure. The following example first defines the procedure (pr1) and then defines the break register (break2) that calls the procedure.

```
*DEFINE PROC prt1 = DO
.*IF x > 0 THEN RETURN true
.*ELSE RETURN false
.*ENDIF
.*END
*
*DEFINE BRKREG break2 = (term, value) call
prt1, #68
```

To run the program using this break register, enter:

```
*GO USING break2
```

Trace Registers

The PSCOPE trace feature displays a trace message when the program it is executing reaches a specified address. The trace message identifies the current execution point, but no break occurs.

The following example defines a trace register named *trace1* that contains tracepoints at statements #80, #124, and at the procedure *error* in the current module:

```
*DEFINE TRCREG trace1 = #80, #124, error
```

A GO command using *trace1* displays a message each time statements #80, #124, or procedure *error* is executed.

On-Line Help

PSCOPE provides on-line help. In addition to obtaining help on topics from a help list, extended versions of PSCOPE error messages can be displayed.

Stepping

With PSCOPE commands users can single-step through high-level-language statements (the LSTEP command), and procedures (the PSTEP command). The LSTEP and PSTEP commands display the statement number of the next high-level-language statement. For example:

```
*LSTEP
[Step at :PAGER#42]
```

The iRMX and XENIX PSCOPE debuggers also have the capability to step through machine instructions (the ISTEP command).

Literally Definitions

LITERALLY definitions are shorthand names for previously defined character strings. They give the user the convenience of being able to customize the debug environment. For example:

```
*DEFINE LITERALLY lit = 'literally'
*DEFINE lit def = 'define'
*def lit stacktop = 'word ss:sp'
```

The Editor

PSCOPE includes an internal editor which provides a subset of Intel's AEDIT text editor's features. With this editor, users can create and modify debug constructs such as patches, debug procedures, and LITERALLY definitions. The internal editor can be used to view source program files on the screen.

THE PSCOPE COMMAND LANGUAGE

The PSCOPE commands are versatile and powerful, yet easy to learn and use. With them, the user can build a high-level environment in which to examine and modify execution of the program under development.

The PSCOPE commands can be divided into functional categories.

Emulation commands instruct PSCOPE-86 to execute the user program. They consist of GO and the two stepping commands, LSTEP, and PSTEP.

Debug environment commands define PATCHes, debug PROCedures, debug *variables*, LITERALLYs, break registers (BRKREG), and trace registers (TRCREG) using the DEFINE command. A user can delete these definitions with the REMOVE command.

Block commands consist of DO-END, COUNT-END, REPEAT-END, and IF-THEN-ELSE constructs. They can be used alone or within debugging procedures and patches.

String functions concatenate strings (CONCAT), return the string length (STRLEN), return a substring (SUBSTR), and accept console input (CI).

Utility commands are general-purpose commands for use in a debugging environment. They consist of the following:

$	is a pseudo-variable that represents the current execution point.
ACTIVE	is a function that determines whether a specified dynamic variable is currently defined on the stack or not.
BASE	sets or displays the current radix.
CALLSTACK	displays the dynamic calling sequence stored on the stack.
DIR	displays all objects of a specified type.
EDIT	invokes the internal, menu-driven, text editor.
EVAL	returns the value of a symbol in binary, decimal, hexadecimal, and ASCII.
EXIT	returns control to the host operating system.
HELP	provides on-line help for selected topics and error messages.
NAMESCOPE	This pseudo-variable represents the current scope of a variable. Gives access to variables without need to use the fully qualified symbolic reference.
OFFSET$OF	is a function that returns the offset of a specified address (virtual or symbolic).
SELECTOR$OF	is a function that returns the selector of a specified address (virtual or symbolic).
WRITE	writes variables and strings to the console's screen.

File handling commands access disk files. The user can load program files to be debugged (LOAD), save patches, debugging procedures, debugging variables, LITERALLYs, and debugging registers in a disk file (PUT and APPEND), read-in these definitions during later debugging sessions (INCLUDE), and record a debugging session in a disk file for later analysis (LIST and NOLIST).

Register access commands provide access to the microprocessor registers and flags. The REGS command displays the registers and the set flags. Users can also inspect or change an individual register by specifying its mnemonic. FLAG represents the flag word. The user can inspect or change each flag separately as a Boolean variable.

ADVANCED PSCOPE COMMANDS FOR iRMX™ AND XENIX OPERATING SYSTEMS

In addition to the basic PSCOPE capabilities, iRMX and XENIX PSCOPE offer complete debugging support for assembly language programs. This support includes the ISTEP single step command, an assembler/disassembler, and register examination and modification for both the main processor and the math coprocessor as follows:

XENIX	80286/80287
iRMX	8086/8087

XENIX PSCOPE also supports debugging of C language programs. Another feature of XENIX PSCOPE allows the user to invoke any XENIX operating system command and return to debugging without altering the state of the debug session.

The Disassembler and Single-Line Assembler

With the disassembler, memory can be displayed as assembly language mnemonics. The next example shows how the ASM command displays the first assembly language instruction that makes up the high-level-language statement #26.

```
*ASM #26
3348H:00CCH FF36000000 PUSH WORD PTR 000H
3348H:00D0H B020       MOV AL,20H
```

An instruction can be changed using the single-line assembler as follows:

```
*ASM 3348H:00D0H = 'MOV AL, 25H'
```

SAMPLE DEBUG SESSION

The following sample debug session illustrates some of PSCOPE'S capabilities. The example program (Figure 1) is written in Pascal. To utilize PSCOPE's symbolic debugging capabilities, the program was compiled with the debug option. After the program is linked and bound (bind option), the executable code can be debugged on PSCOPE.

The following example of the debugging session demonstrates the use of the LITERALLY command to personalize the debug environment. This example assumes PSCOPE has been invoked and the PSCOPE prompt (*) is present on the screen. PSCOPE keywords are in uppercase for the examples, but case is not significant in the PSCOPE command language. No devices or directories will be included with file access commands.

```
*DEFINE LITERALLY d = 'DEFINE'
*d LITERALLY l = 'LITERALLY'
*d l br = 'BRKREG'
*d l tr = 'TRCREG'
```

Next the load time locatable code is loaded, and a directory of the module is requested. In the directory, program symbols are listed with their types. The types corresond to PSCOPE memory object types.

```
*LOAD maxmin.86
*DIR
DIR of:CALC
PQ_OUTPUT ..................TEXT (file)
PA_INPUT ...................TEXT (file)
B ..............................integer
A ..............................integer
SUM ..........................procedure
  X ............................integer
  Y ............................integer
  Z ............................integer
DIFFERENCE ...................procedure
  X ............................integer
  Y ............................integer
  Z ............................integer
MAXMIN .......................procedure
  X ............................integer
  Y ............................integer
```

The next example illustrates using the PSTEP stepping command. PSTEP executes procedures as a single step. When the program requests input, the user must enter values for the stepping to continue. PSCOPE responds with a break message inside brackets ([ ]).

```
*PSTEP
[Step at :CALC#21]
*PSTEP
       Input two integers
[Step at :CALC#22]
                               19 <- user input
                                4 <- user input
[Step at :CALC#23]
*PSTEP
       The sum is 76                 <- note
*PSTEP
[Step at :CALC#24]
       The difference is 15
[Step at :CALC#24]
*PSTEP
       The maximum is  19
       The minimum is   4
```

```
Source File: MAXIN.PAS
Object File: MAXIN.OBJ
Controls Specified: DEBUG

                        (* This program reads two integers and     *)
                        (* determines which is greater.            *)
 STMT  LINE  NESTING  SOURCE TEXT: MAXMIN.PAS
    1     1    0   0  program calc(input,output);
    2     2    0   0  var a,b:integer;

    3     4    0   0  procedure sum(s,y:integer);
    4     5    1   0  var z:integer;
    5     6    1   0  begin
    5     7    1   0    z: = x*y;
    6     8    1   1    writeln('The sum is',z);
    7     9    1   1  end;

    8    11    0   0  procedure difference(s,y:integer);
    9    12    0      var z:integer;
   10    13    1   0  begin
   10    14    1   0    z: = abs(x-y);
   11    15    1   1    writeln("The difference is',z);
   12    16    1   1  end;

   13    18    0   0  procedure maximum(x,y:integer);
   14    19    1   0  begin
   14    20    1   1    if x < y then writeln ('The maximum is',y),
                                              The minimum is',x);
   16    21    1   1    if y < x then writeln ('The maximum is',x,
                                              The minimum is',y);
   18    22    1   1    if x = y then writeln ('The inputs are equivalent')
   20    23    1   1  end;

   21    25    0   0  begin
   21    26    0   1    repeat (*forever*)
   21    27    0   2     write('Input two integers');
   22    28    0   2     readln(a,b);
   23    30    0   2     sum(a,b);
   24    31    0   2     difference(a,b);
   25    32    0   2     maximum(a,b);
   26    33    0   2    until 1 < 0
   27    34    0   2  end.
```

Figure 1. Sample Program

There is a bug in the program. The procedure sum should add the input values, 19 and 4. Instead, it multiplies them. The code on line #5 should specify x+y instead of x*y. With PSCOPE, it is easy to patch this line of code and immediately execute the program without recompiling and relinking.

```
*DEFINE PATCH #5 TIL #6 = z = x+y
```

The next example uses the GO TIL command to illustrate that the patch works.

```
*TO TIL #21
              Input two integers      4
                                     19
              The sum is             23
              The difference is      15
              The maximum is         19
              The minimum is          4
[Break at #21]
```

The final example illustrates the use of a PSCOPE PROC (debug procedure). The PROC is named pr1. Then a BRKREG (break register) is defined to call the procedure. Notice the use of the LITERALLY definitions in the examples.

```
*d PROC pr1 = DO
.*WRITE 'numbers and product are: ',a,b,a,*b
.*WRITE USING ('0,>') 'break?'
.*IF CI = = 'y' THEN RETURN true
..*             ELSE RETURN false
..*ENDIF
.*END
*d br b3 = #21 CALL pr1
*GO USING b3
                Input two integers     23
                                       24
                The sum is             47
                The difference is       1
                The maximum is         24
                The minimum is         23

numbers and the product are: +23 +24 +552
break ? y
[Break at #21]
```

When a debugging session is complete, exit PSCOPE as follows. The debugger will close any open files, write a message to the screen and return to the operating system.

```
*EXIT
PSCOPE terminated
```

BENEFITS

As an interactive, symbolic, high-level language debugger, PSCOPE brings to debugging the same type of productivity enhancements that high-level languages bring to writing software. PSCOPE's benefits are listed below:

- A shortened development cycle. Breakpoints, tracing, and patching decrease the number of edit/compile/link iterations.
- Improved debugging productivity. Since PSCOPE language constructs enable the use of high-level functions and procedures, symbolics and data structures, improvement in debugging productivity is analogous to programming in high-level languages.
- Increased software reliability. Debugging procedures can automate the software testing process.
- Improved project management. Software engineers can debug modules separately. Procedures can be substituted for program stubs.

SPECIFICATIONS

PSCOPE for the iRMX™ Operating System

(for iRMX operating system release 5 or greater on Systems 86/310, 86/330A, and 80/380)

Languages:	•PL/M-86 •Pascal-86 •FORTRAN-86 •ASM86
Documentation:	*PSCOPE-86 High-Level Program Debugger User's Guide* (for iRMX Operating Systems) Order number: 165496
Memory:	110K bytes for iRMX PSCOPE-86 software and buffers

Order Code	Description
iPSC 86 RMX	PSCOPE Program Debugger for the iRMX Operating System

PSCOPE for the XENIX Operating System

(PSCOPE requires an 80286-based system running the Intel XENIX 286 operating system.)

Languages:	Supports any Intel or ISV-supplied 80286 language generating Intel 286 object module format load-time locatable modules. These include: •PL/M •C •FORTRAN •Pascal •ASM
Documentation:	*PSCOPE 286 User's Guide for XENIX Systems,* Order Number: 122281. *PSCOPE 286 Pocket Reference,* Order Number: 122282.
Memory:	286 bytes for XENIX PSCOPE software and buffers.

Order Code	Description
iPSC286XNXSU	Single-user license
iPSC286XNXRO	Incorporation license
iPSC286XNXRF	Royalty fees for incorporations
iPSC286XNXBY	Buy-out license

PSCOPE for the Series III/Series IV Operating System

(PSCOPE-86 runs on an Intellec® system, either stand-alone or in an NDS-II network configuration.)

Languages:	•PL/M 86/88 •Pascal 86/88 •FORTRAN 86/88
Documentation:	*PSCOPE-86 High-Level Program Debugger User's Guide,* Order number 121790
Memory:	The system configuration must include 512K bytes of application memory space. The debugger requires 96K bytes of memory.

Order Code	Description
iMDX-333	for Series III and Series IV
III-951A	I2ICE software for Series III 8″ single density disk drive
III-951B	I2ICE software for Series III 8″ double density disk drive
III-951C	I2ICE for Series IV with 5 1/4″ double density disk drive

An enhanced version of PSCOPE with source-code display is available for the PC-DOS operating system. See Data Sheet order number 280194 for information on PSCOPE-86 for DOS operating systems.

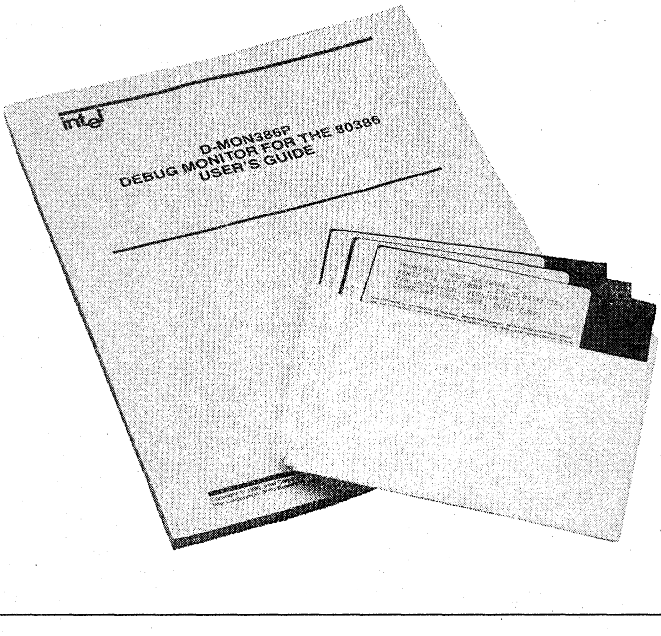
PRELIMINARY

D-MON386P Debug Monitor 386

- Unhosted Monitor Configurable on Any 80386-Based Board
- Provides Program Execution Control, Including Software Breakpoints and Single-Step Execution through Assembly Level Instructions in Both Real and Protected Mode
- Supports Four On-Chip Breakpoints to Recognize Instruction Execution Addresses or Data Access Addresses
- Allows User to Examine and Modify Memory, I/O Ports, and 80386 Registers
- Allows User to Examine Descriptor Tables, Task State Segment, and Page Table
- Allows Disassembly of Memory in 86/286/386 Instruction Mnemonics
- Supports Virtual, Linear, and Physical Addressing
- Supports Real, Protected, and Page Protected Modes of 80386
- Pre-Configured for Intel SBC386/20
- Provides Ten 32-Bit Scratch Registers for Storing Intermediate Values During the Debug Process
- Provides User-Friendly Human Interface with Command Line Editing and Command History

The D-MON386P is an unhosted EPROM based software debug monitor for 80386-based systems and provides system level debug support. D-MON386P does not require a host system for its operation; it can access and control all of the 80386 visible user-hardware resources without any assistance from an operating system. With the help of this monitor, you can set hardware and software breakpoints, examine/modify memory and processor registers, and control program execution. This monitor can be configured to run on any 80386-based target board with a user-supplied communication driver and hardware initialization routine.

The D-MON386P package includes the diskettes that contain the target software to be loaded into the 80386-based target board, and the D-MON386P User's Guide, order number 166186-001.



166187-1

November 1986
Order Number: 166187-001

FUNCTIONAL DESCRIPTION

Overview

D-MON386P provides design aid and debug support for Intel customers who design software to run on the 80386. D-MON386P allows you to perform the following tasks:

- Examine and modify memory, I/O ports, 80386 registers, and display descriptor tables, task state segment, and page table
- Control execution both in real and protected mode
- Set four on-chip breakpoints to recognize instruction execution addresses or data access addresses
- Set software program breakpoints
- Single-step through 80386 instruction execution
- Disassemble program code in 86/286/386 instruction mnemonics
- Evaluate expressions

Processor/Memory Examination and Modification

80386 registers can be accessed mnemonically (e.g. EAX) with the D-MON386P software. Data can be displayed or modified in one of four bases: hexadecimal, decimal, octal, or binary. Program code can be disassembled and displayed as 80386 assembly instruction mnemonics.

Execution Control

The GO command is used to begin execution or to resume execution at the current execution point or at a specified address. The D-MON386P monitor provides the ability to break on execution addresses using either software code patch breaks or hardware debug register breaks provided on the 80386 chip. The software breaks allow breaks on instruction execution only. These breakpoints can be placed on the RAM-based program code. The hardware register breaks allow recognition of the following conditions:

- An instruction boundary
- A data write to a user-specified linear or virtual address
- A data access at a user-specified linear or virtual address

D-MON386P allows you to single-step through your program code by referencing machine-level instructions.

Disassembler

The D-MON386P monitor allows you to disassemble and display user target memory contents in the 80386 assembly instruction mnemonics.

Formatted Displays

The D-MON386P allows you to display all of the 80386 visible hardware resources and pre-defined data structures in easy-to-read formats. These displays include contents of the global descriptor table (GDTs), the local descriptor table (LDTs), the interrupt descriptor tables (IDTs), the task state segments (TSSs), the extended flags register, the segment registers, the 80386 registers, Control Registers (CREGs), page directory, and page tables.

Debugger Command Language

ASM — For disassembling code.

BASE — For changing the default base for command line entry.

COUNT — For repetition of commands or blocks.

DT/GDT/LDT/IDT — For allowing display of descriptor tables.

EVAL — For evaluating expressions.

GO/SWBREAK/SWREMOVE — For controlling program execution.

HELP — For printing all command keywords.

ISTEP — For single-step through machine level user program.

ORD1/ORD2/ORD4/INT1/INT2/INT4 — Data typs used accessing memory.

PORT/DPORT/WPORT — For accessing I/O ports.

REGS — For displaying 80386 registers.

TSS — For accessing the task state segment.

VERSION — For displaying D-MON386P software version being used.

SPECIFICATIONS

Required System Resources

D-MON386P monitor requires exclusive use of the 80386's on-chip debug registers, INT1, and the trap flag.

Target System Environment

The D-MON386P is designed to be configured on any 80386-based target system that provides the following:

- At least 128K bytes of EPROM space
- At least 64K bytes of RAM space
- A serial communications interface

The starting location for the software loaded into the EPROMs and RAM space is user configurable.

Other Required Tools/Systems

The D-MON386P comes pre-configured for Intel SBC386/20 board which uses Intel 8251 and 8254 devices at 1.23 MHz clock rate. Baud rate supported is 9600.

Example drivers are included in User's Guide to aid in writing a custom driver. The following are required to write a custom driver, link it with the monitor code, build the target code, and burn EPROMs:

For writing a custom driver,

- ASM-386
- RLL-386
- Intel System 286/310 running XENIX Rel 3. Update 3 to link driver and build the target code

For burning EPROMs,

- Intel PROM Programmer
- iPPS version 2.1 (hosted on DOS or iNDX)

ORDERING INFORMATION

D-MON386P is a pre-production level release. Production level D-MON386 is scheduled for 1987 and will add support for virtual 86 mode of the 80386. Pre-production level customers who also buy Intel software support contract will be automatically upgraded to the production level product upon release.

Order Code	Description
DMON386P	The D-MON386P monitor package includes the target software diskette containing linkable object code and a user's guide. Single user. (Requires Class I license)
DMON386PCOPY	Right to make additional 8 copies.
DMON386PSRC	D-MON386P source code.
DMON386PRO	D-MON386P. License to incorporate. Class II/Class III license.
DMON386PRF	D-MON386P. Royalty fees.
DMON386PBY	D-MON386. Incorporation fee buyout.

Purchase of the D-MON386P package includes a 90-day support service, which also includes the Software Problem Report Service.

Purchase of any of these options requires the execution of a standard Intel Master Software license. The specific rights granted to users depends on the specific option and the license signed.

Shipping Media

PC-DOS Format	DS/DD 5¼″ diskette

intel®

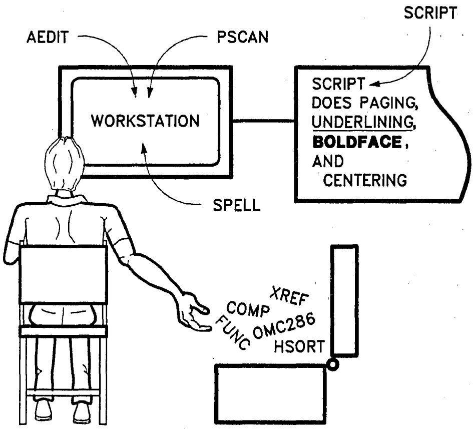
8086 HANDYMAN

8086 SOFTWARE TOOLBOX

- PSCAN Reduces Time Spent Doing Software Entry and Editing
- SCRIPT and SPELL Assist Text Preparation
- OMC286 and E80287 Aid 80286 and 80287 Software Development
- Many Other Valuable 16-Bit Software Tools Are Included
- Runs Under ISIS/iNDX (Series III and IV) and iRMX™86 Operating Systems

AEDIT-86

- Full Screen Editing
- Menu-Driven, Easy To Use
- Powerful Macro Facility
- Dual File Editing
- Split-Screen Windowing
- Automatic File Back-Up
- Runs Under ISIS/iNDX (Series III and IV) and iRMX86 Operating Systems



231364-1

This attractively priced kit of products, AEDIT and the 8086 Software Toolbox, provides the developer with powerful tools for text and code editing, file management, and program development. To find out more about these individual tools please read the data sheets which follow this cover page.

ORDERING INFORMATION

Product Code	Description
MDX374	8086 Handyman under ISIS/iNDX Operating System
RMX86HAN	8086 Handyman under iRMX86 Operating System

March 1985
Order Number: 231364-001

8086 SOFTWARE TOOLBOX

- **Collection of Tools That Speed Software Development**
- **MPL, a Standalone Macro Processor, is Ideal for Debugging Macros**
- **PSCAN Reduces Time Spent Doing Software Entry and Editing**
- **SCRIPT and SPELL Assist Text Preparation**
- **OMC286 and E80287 Aid 80286 and 80287 Software Development**
- **Many Other Valuable 16-Bit Software Tools are Included**
- **Runs on Series III and Series IV Microcomputer Development Systems**
- **Runs Under iRMX™86 Operating System**

The 8086 Software Toolbox is a collection of 16-bit software tools that can significantly improve programmer productivity. These tools are valuable for text formatting, editing, and preparation, software testing and performance analysis, 286/287 software development, and a multitude of other applications.

Text processing tools ease document formatting and preparation. PSCAN is a syntax-scanning editor for the PL/M language. It catches syntax errors in the editing stage and provides automatic formatting of PL/M code and more. SCRIPT is a text formatting program that uses commands embedded in text to do paging, centering, left and right margins, subscripts, etc. SPELL finds misspelled words in a text file and comes with a user expandable dictionary. COMP prepares two text or source files and displays their differences.

Test and performance analysis tools aid software testing and performance evaluation. PERF, a performance analysis tool for 8086 software, ideal for isolating code "hot spots." PASSIF is a general-purpose assertion checking and reporting tool perfect for running test suites.

Software development for 286/287 components is assisted by two software tools: OMC286, an 8086 to 80286 object module convertor, and E80287, an 80287 emulator that runs on the 80286.

Additional tools are included that aid 16-bit software development efforts.

Text Editing and Processing
PSCAN SCRIPT MPL SPELL WSORT

Performance Measurement & Testing
PERF GRAFIT PASSIF

286/287 Development
OMC286 E80287

Miscellaneous Tools
COMP FUNC XREF DC HSORT ESORT

8086 SOFTWARE TOOLBOX TOOLS

FUNCTIONAL DESCRIPTION

Text Editing and Processing

PSCAN—syntax scanning editor that supports all the functions of AEDIT-86 Release 1.0 plus specialized functions for entering and editing PL/M source programs. PSCAN verifies correct code entry as you type, suppressing time consuming recompilations. In addition, PSCAN provides facilities to automatically format PL/M code, and can perform editor functions on statements, blocks or procedures.

SCRIPT—text formatting program that does paging, centering, left and right margins, justification, page headers and footers, underlines, boldface type, subscripts and superscripts, upper and lower case, and much more. Formatting commands are embedded in text.

MPL—standalone macro processor that processes the macro language used in 8086, 80286, 8089, and 8051 assemblers. Can be used interactively which makes it ideal for debugging macros. MPL can be used to preprocess any text file.

SPELL—finds misspelled words in a text file. Dictionary of correctly spelled words is user expandable.

WSORT—utility for creating the SPELL dictionary.

COMP—performs line-oriented text file comparison (shows source changes). Also understands 8086 object module formats for comparing 8086 object files.

Performance Measurement and Testing

PASSIF—general-purpose assertion checking, testing, and reporting tool. Helps automate the software testing process.

PERF—performance analysis tool for 8086 software. Monitors references in the code segment; segment monitored is user defined. Works with small or compact bound loadable modules. Ideal for isolating code "hot spots." Will only run on the Series III.

GRAFIT—graphing utility for use with PERF.

Miscellaneous Tools

OMC286—object module convertor that converts 8086 object modules into 80286 object modules.

E80287—an 80287 emulator that runs on the 80286.

FFUNC—allows user to redefine the keys on a Series III keyboard and define function keys. Requires the iMDX 511 firmware.

XREF—produces cross-reference tables from translator list files. Cross-references all symbols—variables, labels, literallys, and quoted strings.

DC—floating point desk calculator program; allows variable definitions.

HSORT—in memory heap soft utility.

ESORT—very flexible sort program.

SPECIFICATIONS

Operating Environment

ISIS Operating System with RUN or INDX Operating System executing on Series III or Series IV Microcomputer Development Systems.

iRMX86 Operating System executing in SYS X86/3XX environment.

Documentation

"8086 Software Toolbox"
(122203)

Software Support

This product includes a 90-day initial support consisting of new software releases, updates, subscription services (software performance reports and technical reports), and telephone hotline support. Additional software support services are available separately.

ORDERING INFORMATION

Product Code	Description
MDX 374	Handyman Kit, including 8086 Software Toolbox, running on ISIS/iNDX
RMX86 TLB	8086 Software Toolbox under iRMX™86
RMX86 HAN	Handyman Kit, including 8086 Software Toolbox, under iRMX™86

AEDIT TEXT EDITOR

- **AEDIT-80 Operates on Any Intellec® Series II, Model 800 or iPDS™ Development System**
- **Full Screen Editing**
- **Menu-Driven, Easy To Use**
- **Easy Handling of Large Blocks of Text**
- **Dual File Editing**
- **AEDIT-86 Operates on Any Intellec® Series III, Series IV, or iRMX™ System**
- **Powerful Macro Facility**
- **Split-Screen Windowing**
- **Designed for the Programmer and Technical Writer**

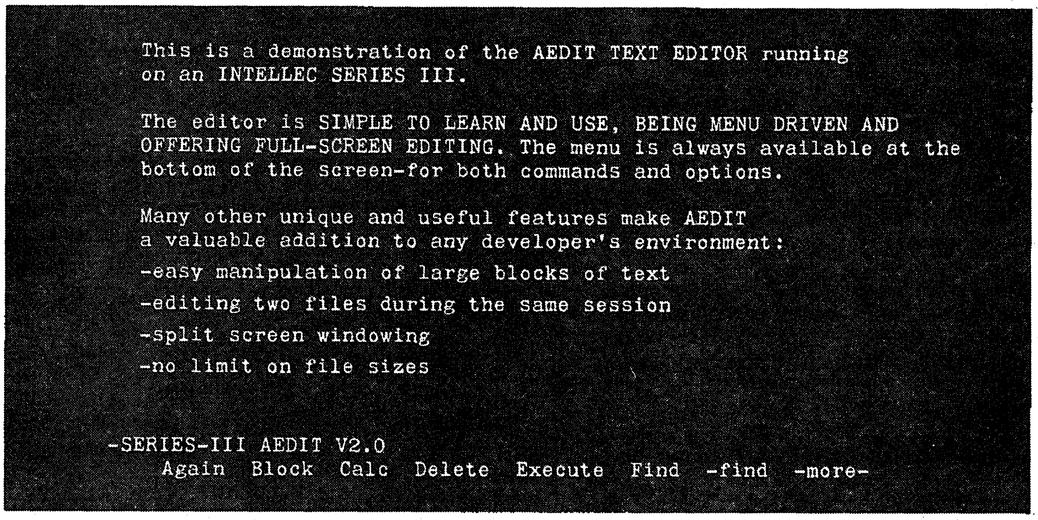
AEDIT is a full screen editor for use on any Intellec® Development or iRMX™ system. It is designed to be easy to learn and easy to use. At all times the user is guided by a menu which is used not only to select commands, but also to select options to commands. There is no need to constantly refer to or memorize detailed manuals.

AEDIT provides full screen editing capabilities and offers features to easily handle (move, copy, delete) large blocks of text. In addition to the basic editing abilities, AEDIT supports tagging positions in the text, string search and replace commands, and the option of automatic text identation, spilling, and formatting. AEDIT is able to edit files of any length and optionally creates back-up copies of the file being edited.

With AEDIT, two files can be edited during one session. The user can easily switch between the files for quick reference, editing, or to transfer text from one file to the other. Using the windowing capabilities available with AEDIT-86, both of these files may be displayed simultaneously in a split-screen format.

AEDIT supports a powerful macro facility. AEDIT can create macros by simply keeping track of what a user is executing, "learning" the function the macro is to perform. The editor remembers the user's actions for later execution, and can store them in a file if requested. Alternatively, a user may enter a macro using AEDIT's macro language, or modify any existing macro interactively.

These and many other features combine to make AEDIT the editor of choice.



```
    This is a demonstration of the AEDIT TEXT EDITOR running
    on an INTELLEC SERIES III.

    The editor is SIMPLE TO LEARN AND USE, BEING MENU DRIVEN AND
    OFFERING FULL-SCREEN EDITING. The menu is always available at the
    bottom of the screen-for both commands and options.

    Many other unique and useful features make AEDIT
    a valuable addition to any developer's environment:
    -easy manipulation of large blocks of text
    -editing two files during the same session
    -split screen windowing
    -no limit on file sizes


 -SERIES-III AEDIT V2.0
      Again  Block  Calc  Delete  Execute  Find  -find  -more-
```

231364-2

MANUALS

AEDIT is supplied with a user manual documenting all the aspects of the editor, and a pocket reference card. The manual includes an introductory tutorial.

HOST SYSTEM

AEDIT-80 is an 8080/8085-based utility and can be run on any Intellec Development System, Series IIE, Series II, Model 800, or IPDS, as well as on ISIS Cluster workstations.

The higher-performance AEDIT-86 is an 8086-based utility that can be run on any Intellec Series IIIE, Series III, or Series IV Development system. Any Series IIE, Series II or Model 800 system can be upgraded to Series III functionality. AEDIT-86 is also available for the iRMX Operating Systems.

AEDIT can be configured to run with non-Intel terminals. Tested configurations are available for the following popular terminals:

ADDS Regent 200, Viewpoint 3A+
Beehive Mini-Bee
DEC VT52, VT100
Hazeltine 1510
Lear-Seigler ADM-3A
Zentec ZMS-35

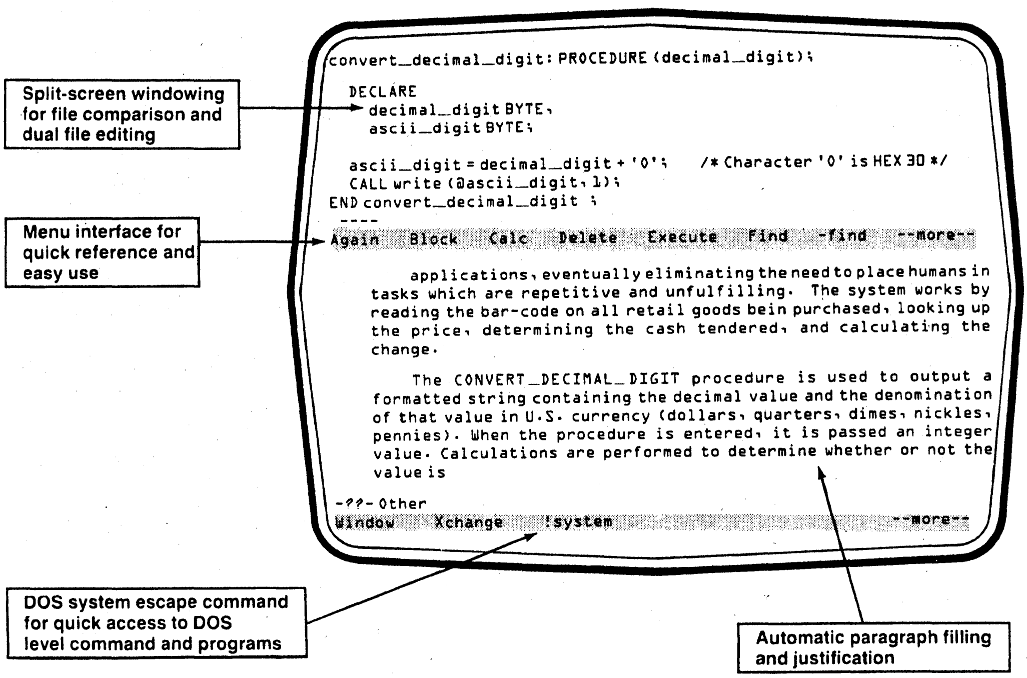
Regent 200 is a trademark of ADDS
Mini-Bee is a trademark of Beehive
DEC designated Digital Equipment Corporation
ADM-3A is a trademark of Lear-Siegler

ORDERING INFORMATION

Product Code	Description
iMDX-335	AEDIT-80 Text Editor. Includes 8″ single and double-density diskettes for Series IIE, Series II, or Model 800, and a 5¼″ diskette for iPDS.
iMDX-334	AEDIT-86 Text Editor. Includes 8″ single and double density diskettes for Series III and a 5¼″ double density diskette for the Series IV.
iRMX864	AEDIT-86 Text Editor under iRMX86 includes 5¼″ and 8″ double density RMX format diskettes.

AEDIT
SOURCE CODE AND TEXT EDITOR FOR PC-DOS

- **A Full-Screen Source Code Editing and Documentation Tool Designed Specifically for Software Programmers and Technical Writers**
- **Complete Range of Editing Support— from Document Processing to HEX Code Entry and Modification**
- **Supports System Escape for Quick Execution of PC-DOS System Level Commands**
- **Full Macro Support for Complex or Repetitive Editing Tasks**
- **Dual File Support with Optional Split-Screen Windowing**
- **No Limit to File Size or Line Length**
- **Quick Response with an Easy to Use, Menu-Driven Interface**
- **Configurable and Extensible for Complete Control of the Editing Process**


Split-screen windowing for file comparison and dual file editing
Menu interface for quick reference and easy use
DOS system escape command for quick access to DOS level command and programs
Automatic paragraph filling and justification

```
convert_decimal_digit: PROCEDURE (decimal_digit);
  DECLARE
    decimal_digit BYTE,
    ascii_digit BYTE;

  ascii_digit = decimal_digit + '0';    /* Character '0' is HEX 30 */
  CALL write (@ascii_digit, 1);
END convert_decimal_digit ;
 ----
Again   Block   Calc   Delete   Execute   Find   -find   --more--

         applications, eventually eliminating the need to place humans in
    tasks which are repetitive and unfulfilling. The system works by
    reading the bar-code on all retail goods bein purchased, looking up
    the price, determining the cash tendered, and calculating the
    change.

         The CONVERT_DECIMAL_DIGIT procedure is used to output a
    formatted string containing the decimal value and the denomination
    of that value in U.S. currency (dollars, quarters, dimes, nickles,
    pennies). When the procedure is entered, it is passed an integer
    value. Calculations are performed to determine whether or not the
    value is

-??- Other
Window   Xchange   !system                                   --more--
```

280170–1

* IBM is a registered trademark of International Business Machines.

October 1986
Order Number: 280170-001

PRODUCT FEATURES

Programmer Support

AEDIT is a full-screen text editing system designed specifically for software engineers and technical writers. With the facilities for automatic program block indentation, HEX display and input, and full macro support, AEDIT is an essential tool for any programming environment. And with AEDIT, the output file is the pure ASCII text (or HEX code) you input—no special characters or proprietary formats.

Dual file editing means you can create source code and its supporting documentation at the same time. Keep your program listing with its errors in the background for easy reference while correcting the source in the foreground. Using the split-screen windowing capability, it is easy to compare two files, or copy text from one to the other. The DOS system-escape command eliminates the need to leave the editor to compile a program, get a directory listing, or execute any other program executable at the DOS system level.

There are no limits placed on the size of the file or the length of the lines processed with AEDIT. It even has a batch mode for those times when you need to make automatic string substitutions or insertions in a number of separate text files.

Powerful Text Editor

As a text editor, AEDIT is versatile and complete. In addition to simple character insertion and cursor positioning commands, AEDIT supports a number of text block processing commands. Using these commands you will be able to easily move, copy, or delete both small and large blocks of text. AEDIT also provides facilities for forward or reverse string searches, string replacement and query-replace.

AEDIT removes the restriction of only inserting characters when adding or modifying text. When adding text with AEDIT you may choose to either insert characters at the current cursor location, or overwrite the existing text as you type. This flexibility simplifies the creation and editing of tables and charts.

User Interface

The menu-driven interface AEDIT provides makes it unnecessary to memorize long lists of commands and their syntax. Instead, a complete list of the commands or options available at any point is always displayed at the bottom of the screen. This makes AEDIT both easy to learn and easy to use.

Full Flexibility

In addition to the standard PC terminal support provided with AEDIT, you are able to configure AEDIT to work with almost any terminal. This, along with user-definable macros and fully adjustable tabs, margins, and case sensitivity, combine to make AEDIT one of the most flexible editors available today.

Macro Support

AEDIT will create macros by simply keeping track of the commands and text you type, "learning" the function the macro is to perform. The editor remembers your actions for later execution, or you may store them in a file to use in a later editing session.

Alternatively, you can design a macro using AEDIT's powerful macro language. Included with the editor is an extensive library of useful macros which you may use or modify to meet your individual editing needs.

Text Processing

For your documentation needs, paragraph filling or justification simplifies the chore of document formatting. Automatic carriage return insertion means you can focus on the content of what you are typing instead of how close you are to the edge of the screen.

DOCUMENTATION

AEDIT is supplied with a complete user manual (order number: 122717) that documents all aspects of the editor and includes an introductory tutorial. Additionally, a pocket reference (order number 122721) is provided for quick command look-up.

HOST SYSTEM

AEDIT for PC-DOS has been designed to run on the IBM* PC XT, IBM PC AT and compatibles. It has been tested and evaluated for the PC-DOS 3.0 (or greater) operating system.

Versions of AEDIT are available for the Intel Intellec® Series-II, Series III, and Series IV development workstations, as well as the iRMX™ 86 and iRMX 286 operating systems.

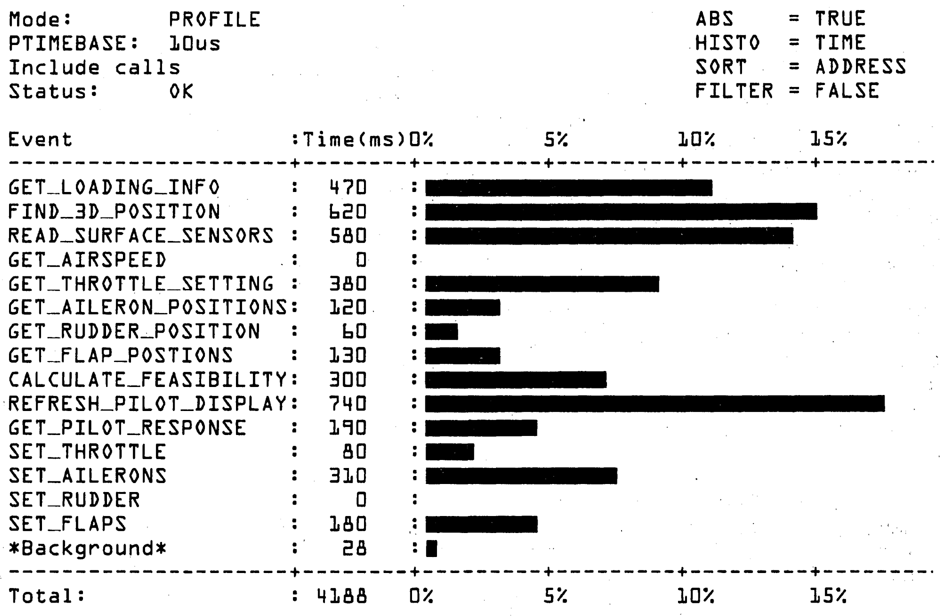
ORDERING INFORMATION

Order Code	Description
D86EDIEU	AEDIT Source Code Editor Release 2.2 for PC-DOS with supporting documentation
122717	AEDIT-DOS User's Guide
122721	AEDIT-DOS Pocket Reference

iPAT™ PERFORMANCE ANALYSIS TOOL

- **Provides Real-Time Performance Analysis and Real-Time Test Coverage of Code Written for 8086/8088, 80186/80188, and 80286 Processors**
- **Displays Performance-Analysis Histograms to Isolate Slow Code**
- **Displays Test Coverage Tables to Isolate Untested Code; Permits Saving and Updating Test Results**
- **Measures Interrupt Latency**
- **Does not Intrude Into Program Being Analyzed**
- **Collects 100% of Execution Data**
- **Complements Emulator by Allowing Simultaneous Debugging and Performance Analysis**
- **Permits Activation of Analysis using Emulator Procedures**
- **Handles Up to 24-Bit Execution Address Space**
- **Permits Specification of Analysis Address Ranges Symbolically or with Absolute Addresses**
- **Provides Flexible Isolation of Code Ranges, Windowed Events, and Interrupt Activity**

The Intel Performance Analysis Tool (iPAT™) helps software engineers optimize code and improve software reliability. Software object code generated by Intel assemblers and Intel compilers (e.g., for C, PL/M, Pascal, Ada, and FORTRAN) can be analyzed symbolically to improve software execution efficiency and to validate test coverage. Any object code that lacks Intel compiler information—but that can be run by Intel emulators and for which an absolute program map is available—can also be analyzed (nonsymbolically) by the iPAT analyst. iPAT operation is currently supported via a target interface to the I^{2}ICE™ Integrated Instrumentation and In-Circuit Emulation System.



```
Mode:        PROFILE                                   ABS    = TRUE
PTIMEBASE:   10us                                      HISTO  = TIME
Include calls                                          SORT   = ADDRESS
Status:      OK                                        FILTER = FALSE

Event                   :Time(ms)0%        5%         10%        15%
------------------------+--------+---------+---------+---------+--------
GET_LOADING_INFO        :  470   :
FIND_3D_POSITION        :  620   :
READ_SURFACE_SENSORS    :  580   :
GET_AIRSPEED            :    0   :
GET_THROTTLE_SETTING    :  380   :
GET_AILERON_POSITIONS   :  120   :
GET_RUDDER_POSITION     :   60   :
GET_FLAP_POSTIONS       :  130   :
CALCULATE_FEASIBILITY   :  300   :
REFRESH_PILOT_DISPLAY   :  740   :
GET_PILOT_RESPONSE      :  190   :
SET_THROTTLE            :   80   :
SET_AILERONS            :  310   :
SET_RUDDER              :    0   :
SET_FLAPS               :  180   :
*Background*            :   28   :
------------------------+--------+---------+---------+---------+--------
Total:                  : 4188    0%        5%         10%        15%
```

280165-1

October 1986
Order Number: 280165-001

PERFORMANCE ANALYSIS INTRODUCTION

The size and complexity of software has increased with each new generation of microprocessors. As a result, it has become increasingly important to optimize software and to ensure its reliability. The iPAT analyst answers these needs.

Optimizing Software

Optimizing software means maximizing software speed without sacrificing functionality or reliability. To increase speed, execution bottlenecks need careful attention. But, how can the crucial slow code be located?

Without the iPAT analyst, you might analyze the various paths in the source code and make educated guesses where the bottlenecks will occur. Or you might place count statements in the code to learn how often the various paths are entered. Neither of these methods can ensure that you really will isolate the bottlenecks. Furthermore, the second method is intrusive—with the extra statements, real-time operation of your original code cannot occur.

The iPAT analyst provides the solution to the software engineering problem of locating crucial code. With the iPAT analyst, you can quickly and easily show (with histograms or tables) timing and count information for specified program modules, procedures, lines, or absolute address ranges. Because it fully supports symbolic information from Intel high-level languages, the iPAT analyst enables you to use the names of procedures and modules to specify ranges that you want to analyze. (For object code that lacks symbolic information, consult your code's absolute program map and then specify absolute address ranges of interest.)

Furthermore, the iPAT analyst is nonintrusive and operates in real-time. It does not sample program operation on a statistical basis; rather, it has available to it each address that is executed so that no potentially troublesome code will be overlooked. (The iPAT analyst can also monitor when interrupts occur.)

Software teams currently doing their coding in assembly language (to ensure speed of program execution) can now consider writing future code in high-level languages. Since much code does not have a significant effect on overall program speed, after the code is written in high-level language, the bottlenecks can be located by the iPAT analyst. Then, if need be, the code causing the bottlenecks can be redone in assembly language. This method of software development means faster product development, since coding can progress much faster using a high-level language.

Measuring Hardware-Interrupt-to-Software-Response Time (Latency)

The iPAT analyst not only allows you to acquire timing and count information on software events; it also allows you to examine hardware-interrupt-to-software interactions. For example, you can measure how long it is before the appropriate service routine is executed in response to a hardware interrupt. If the measured hardware-interrupt-to-software latency period is not acceptable, the iPAT analyst can help you isolate the causes.

Coordinating Performance Analysis with Emulator Controls

Using the emulator with the iPAT analyst also enables you to analyze program execution as a function of differing target-system conditions. You can set up the conditions in the target system with the emulator, set up iPAT data collection for a section of code, then run the program with the iPAT analyst activated. Change the target conditions and repeat program execution and performance analysis.

You can also create emulator procedures (PROCs) containing emulator commands that trigger performance analysis as a function of selected software or hardware events.

Ensuring Software Reliability

As code is developed, there is a need to ensure that it has no defective code. Typically for this purpose, test suites are developed by software engineers. The engineers use their theoretical understanding of the software to devise test suites that will exercise the code paths. Then, the program under test is run with the test suites, and the program's output is examined. If the desired values are present in the output, it is assumed that the paths were tested. But this is an inference; the test results do not themselves show whether the paths were all exercised.

Thus, without the help of the iPAT analyst, testers cannot be confident that their tests exercised all the code. As a result, there may be a tendency to restrict designs to familiar algorithms and techniques, so that previously successful test suites can be reused.

By contrast, the coverage mode in the iPAT analyst enables you to identify easily and quickly which lines or procedures in your software are not being exercised by the test suites. Thus, you need not restrict your test suites or your coding techniques and options. Furthermore, when the iPAT analyst reveals untested code, you can modify your test suites until the iPAT analyst shows that all code is tested.

How the iPAT™ Analyst Affects Development

As your code is being developed, preliminary analyses can be made with the iPAT analyst. Then, when your system hardware is developed to the point that code can be loaded into it and run, the iPAT analyst can make real-time measurements. Refinements of software and test suites can occur up until product release, with each new modification being checked by the iPAT analyst for execution efficiency and reliability.

But, the iPAT analyst's usefulness to the product is not at an end, because most products are enhanced after the first release. As new releases are being prepared (to add new features), the iPAT analyst will be available to analyze the new code and the newest test suites.

The iPAT analyst can also be used to enhance existing products—products that were developed before performance analysis was available. you can examine existing code with the iPAT analyst to identify slow code; recode; re-examine; then, when performance (and reliability) have been improved, release the enhanced products.

The iPAT analyst provides a way for software engineers to check whether the software meets performance specifications. In addition, in the future you will be able to write more meaningful specifications that cite desired iPAT measurements.

If portions of code are likely to be reused, the iPAT analyst can provide measurements of the reusable code's performance characteristics. Then, future users of the code will know in advance what to expect from the code.

Another use of performance analysis is encouraging engineers to engage in "what-if" thinking. They can ask, "What if this portion of the code was designed this way?" Then, after they complete several ways of coding, the various versions can be analyzed by the iPAT analyst to reveal which has the greatest efficiency.

PHYSICAL DESCRIPTION

The iPAT system consists of hardware and software.

Figure 1 shows the iPAT hardware connected to the I^{2}ICE emulation system and hosted by an IBM PC AT. The iPAT hardware includes the following:

- Power supply (with AC and DC power cables)
- Core module
- Emulator-specific target interface (which enables the core module to function with a specific emulator)
- Cable for connecting the core module to the target interface
- RS-232 serial cable for connecting the core module to the host system

iPAT software is integrated with the emulator software. Thus, with the iPAT/I^{2}ICE system target interface you receive I^{2}ICE system host software. (You do not receive I^{2}ICE system probe software; continue to use the probe software—version 1.7 or later—supplied with the I^{2}ICE system.) In addition, you receive iPAT diagnostic and tutorial software.

FUNCTIONAL DESCRIPTION

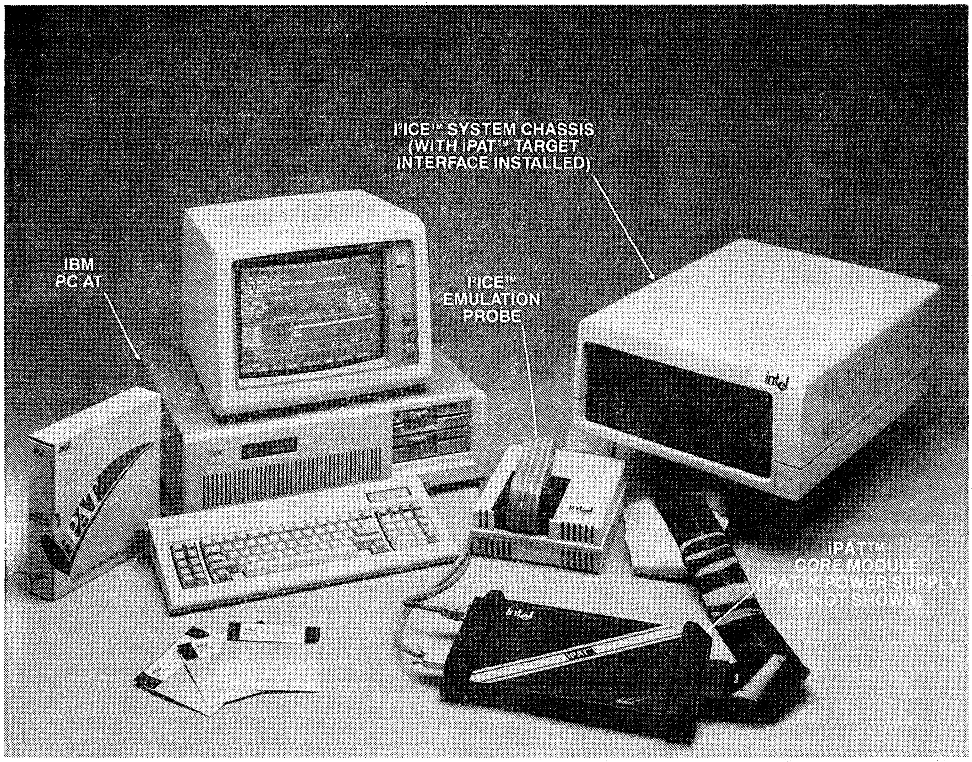
Users will begin analysis of their code by obtaining an overview of their software's operation, and then restrict their focus as they home in on the problem areas in their code. Five analysis modes are available:

- profile
- coverage
- windowed event count
- duration
- linkage

Of these, the profile and coverage modes can be used to acquire both overviews and more localized inspection of your software behavior. The iPAT windowed-event-count, duration, and linkage modes each provide specific perspectives on localized software behavior.

GAINING AN OVERVIEW OF SOFTWARE OPERATION

Gaining an overview of your software operation is simple with the iPAT analyst. If you want an overview of program activity, you load your program, select


I²ICE™ SYSTEM CHASSIS (WITH iPAT™ TARGET INTERFACE INSTALLED)
IBM PC AT
I²ICE™ EMULATION PROBE
iPAT™ CORE MODULE (iPAT™ POWER SUPPLY IS NOT SHOWN)

280165-2

Figure 1. The iPAT™ Analyst Used with an IBM PC AT

the profile analysis mode, and then run the program. To do so, you need only enter the following commands:

```
LOAD new_program
PAT INIT PROFILE
GO
```

To display the results (during or after program execution), enter:

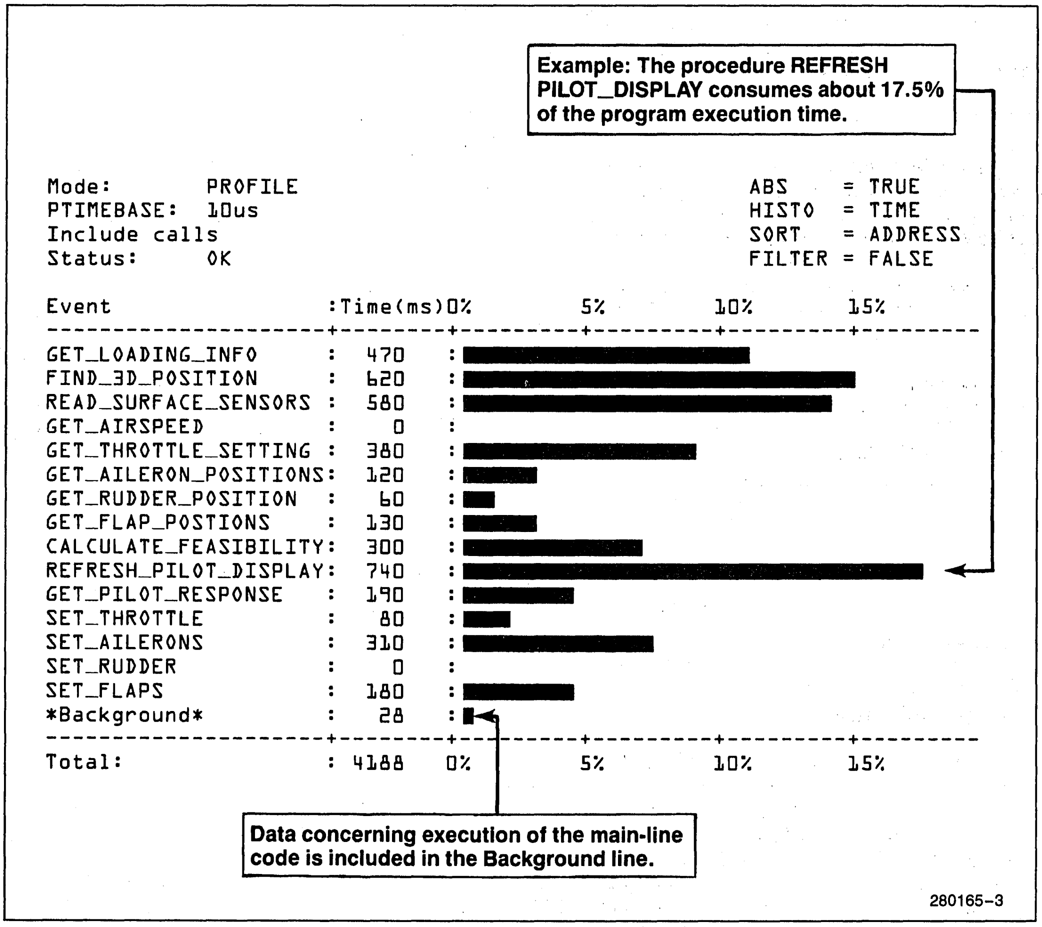
```
PAT DISPLAY
```

iPAT options and controls provide considerable flexibility in monitoring and displaying information about your code. Yet the default settings have been designed with a view to typical applications and ease-of-learning. Default operation in the profile mode monitors all procedures in the user program and measures their real-time characteristics.

The default display for profile mode is a histogram that shows the time spent in each of your program's procedures. See Figure 2 for a sample default profile display.

Acquiring an overview of test coverage is also simple. First set up the coverage mode.

```
PAT INIT COVERAGE
```



Example: The procedure REFRESH PILOT_DISPLAY consumes about 17.5% of the program execution time.

```
Mode:        PROFILE                                       ABS    = TRUE
PTIMEBASE:   10us                                          HISTO  = TIME
Include calls                                              SORT   = ADDRESS
Status:      OK                                            FILTER = FALSE

Event                 :Time(ms)0%        5%        10%        15%
----------------------+--------+---------+---------+---------+---------
GET_LOADING_INFO      :  470   :
FIND_3D_POSITION      :  620   :
READ_SURFACE_SENSORS  :  580   :
GET_AIRSPEED          :    0   :
GET_THROTTLE_SETTING  :  380   :
GET_AILERON_POSITIONS :  120   :
GET_RUDDER_POSITION   :   60   :
GET_FLAP_POSTIONS     :  130   :
CALCULATE_FEASIBILITY :  300   :
REFRESH_PILOT_DISPLAY :  740   :
GET_PILOT_RESPONSE    :  190   :
SET_THROTTLE          :   80   :
SET_AILERONS          :  310   :
SET_RUDDER            :    0   :
SET_FLAPS             :  180   :
*Background*          :   28   :
----------------------+--------+---------+---------+---------+---------
Total:                : 4188    0%       5%        10%        15%
```

Data concerning execution of the main-line code is included in the Background line.

280165–3

Figure 2. Profile Mode: Time Histogram Display

Then, run your program with the data inputs from your tests suites, and request a display of results using the following commands:

```
GO FROM top
PAT DISPLAY
```

By default, the coverage display lists all procedures and indicates whether each was executed. Figure 3 shows a sample coverage display. It indicates that no code in the procedures GET_AIR_SPEED and SET_RUDDER was executed by the test suites.

GETTING OTHER VIEWS OF SOFTWARE OPERATION

To obtain more refined information about program operation and test coverage, you can use all five analysis modes. For all modes, the basic display command is the same:

```
PAT DISPLAY
```

You can select whether the display should be renewed periodically during real-time program execution. If you select periodic renewal, you can also select how frequently (in seconds) it is renewed.

Data collection occurs with one of five selectable time bases: 100 μs, 10 μs, 1 μs, and 200 ns. The default value is 10 μs.

The following sections describe how each of the five analysis modes and their associated displays can be used to obtain other kinds of overviews and how to localize the collection of data.

Coverage Mode

The default features of the coverage mode have already been described. Once you have a coverage overview, you may want to restrict the data displayed.

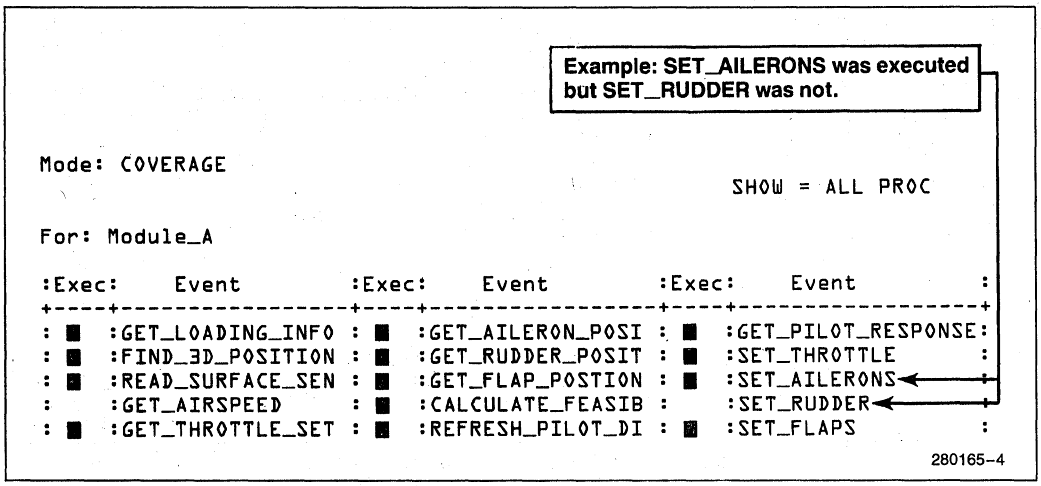
For example, if the default coverage information shows that all procedures were executed by test suites, you may next wish to determine whether all lines in certain procedures were executed. You would then request a display (for the address range desired) of the lines not executed. Using this method, you can obtain very refined test-coverage information and thus help ensure software reliability.

Profile Mode

For profile mode there are a number of ways you can control analysis and the display of data.

Profile-Mode Analysis: For profile mode, data, by default, is collected on program procedures. If you want to acquire an even wider overview, you can change the focus to program modules. Or, for a very close view, you can request that data be collected on the lines executed.

After you have examined your program's profile display, you may notice that several procedures are using excessive time. You will next want to use the iPAT analyst to determine whether the time spent is really attributable to those procedures or rather to calls by those procedures to other procedures. In the default case, when a procedure calls another, the time spent in the called procedure is accumulat-



Example: SET_AILERONS was executed but SET_RUDDER was not.

```
Mode: COVERAGE
                                                    SHOW = ALL PROC

For: Module_A

:Exec:    Event          :Exec:    Event          :Exec:    Event             :
+----+-----------------+----+-----------------+----+-------------------+
: ■  :GET_LOADING_INFO  : ■  :GET_AILERON_POSI  : ■  :GET_PILOT_RESPONSE:
: ■  :FIND_3D_POSITION  : ■  :GET_RUDDER_POSIT  : ■  :SET_THROTTLE       :
: ■  :READ_SURFACE_SEN  : ■  :GET_FLAP_POSTION  : ■  :SET_AILERONS       :
:    :GET_AIRSPEED      : ■  :CALCULATE_FEASIB  :    :SET_RUDDER         :
: ■  :GET_THROTTLE_SET  : ■  :REFRESH_PILOT_DI  : ■  :SET_FLAPS          :
```

280165-4

Figure 3. Coverage Mode: Display Showing Procedures Executed and Not Executed

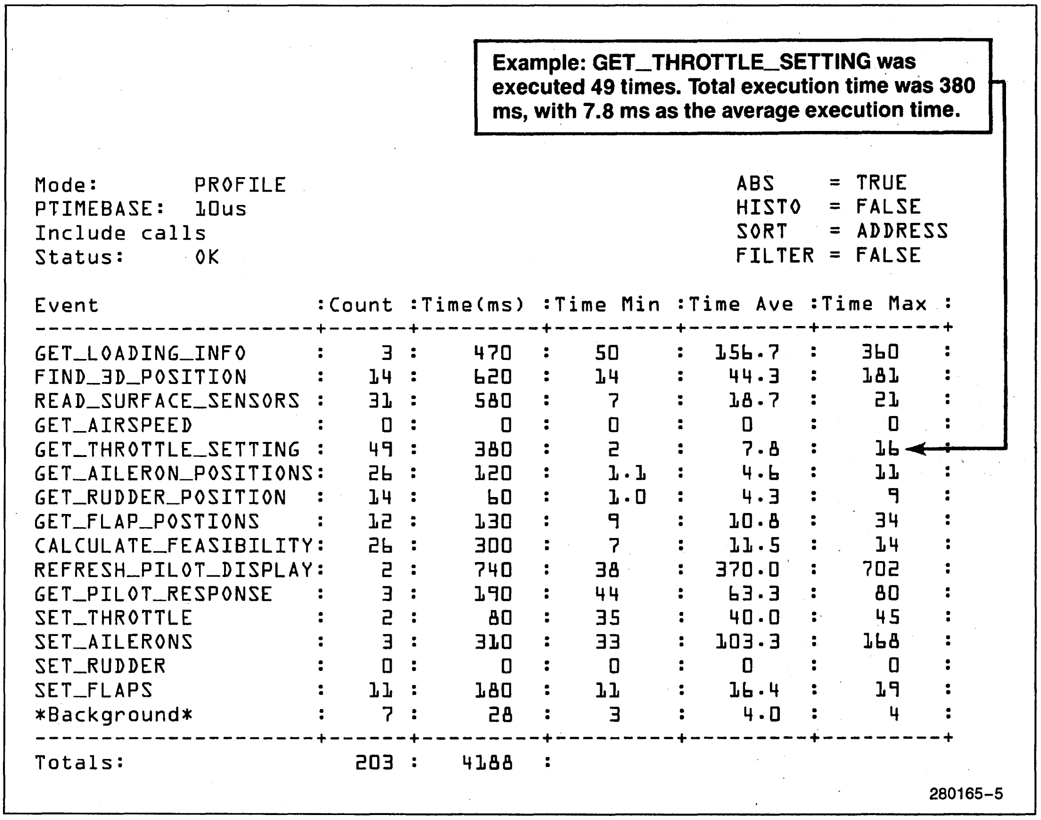
ed by the iPAT analyst as part of the calling procedure's time. If you do not want time charged to the caller, change the control so that the time accumulated by calling procedures excludes time used by called procedures. Then rerun the program and collect new data. Now, by comparing the time charged to the calling procedure in the two cases, you can determine to what extent calls by the procedure use excessive time.

When you use profile mode, you need not collect data on the whole program. You can restrict the range of modules, procedures, or lines that are profiled. In addition, you can restrict the profile to specified absolute-address ranges or to an interrupt-address pair.

Profile-Mode Displays: The default profile display (shown in Figure 2) provides a histogram of the time used by program procedures. Once you notice that some procedures are taking too long, you will want to determine how often those procedures are called. Is the excessive time a result of their being called frequently or the result of slow code? To find out, you need only select a display of count information. A histogram appears immediately (derived from already-acquired data). In the histogram, the lines for the procedures that are taking too long will show whether their counts are small (implying slow code) or large.

You can also display count and time information simultaneously by selecting the table display option. To do so, simply change the HISTO control to false and request a new display. Figure 4 shows a sample profile table display.

Another display control allows you to specify in what order data is presented. By default, data is presented in address order. But you can also direct the iPAT analyst to arrange results in time order or count order, with highest values first.


Example: GET_THROTTLE_SETTING was executed 49 times. Total execution time was 380 ms, with 7.8 ms as the average execution time.

```
Mode:          PROFILE                                      ABS    = TRUE
PTIMEBASE:     10us                                         HISTO  = FALSE
Include calls                                               SORT   = ADDRESS
Status:        OK                                           FILTER = FALSE

Event                   :Count :Time(ms) :Time Min :Time Ave :Time Max :
------------------------+------+---------+---------+---------+---------+
GET_LOADING_INFO        :    3 :     470 :    50   :  156.7  :    360  :
FIND_3D_POSITION        :   14 :     620 :    14   :   44.3  :    181  :
READ_SURFACE_SENSORS    :   31 :     580 :     7   :   18.7  :     21  :
GET_AIRSPEED            :    0 :       0 :     0   :    0    :      0  :
GET_THROTTLE_SETTING    :   49 :     380 :     2   :    7.8  :     16  :
GET_AILERON_POSITIONS   :   26 :     120 :    1.1  :    4.6  :     11  :
GET_RUDDER_POSITION     :   14 :      60 :    1.0  :    4.3  :      9  :
GET_FLAP_POSTIONS       :   12 :     130 :     9   :   10.8  :     34  :
CALCULATE_FEASIBILITY   :   26 :     300 :     7   :   11.5  :     14  :
REFRESH_PILOT_DISPLAY   :    2 :     740 :    38   :  370.0  :    702  :
GET_PILOT_RESPONSE      :    3 :     190 :    44   :   63.3  :     80  :
SET_THROTTLE            :    2 :      80 :    35   :   40.0  :     45  :
SET_AILERONS            :    3 :     310 :    33   :  103.3  :    168  :
SET_RUDDER              :    0 :       0 :     0   :    0    :      0  :
SET_FLAPS               :   11 :     180 :    11   :   16.4  :     19  :
*Background*            :    7 :      28 :     3   :    4.0  :      4  :
------------------------+------+---------+---------+---------+---------+
Totals:                    203 :    4188 :
```

Figure 4. Profile Mode: Table Display

Duration Mode

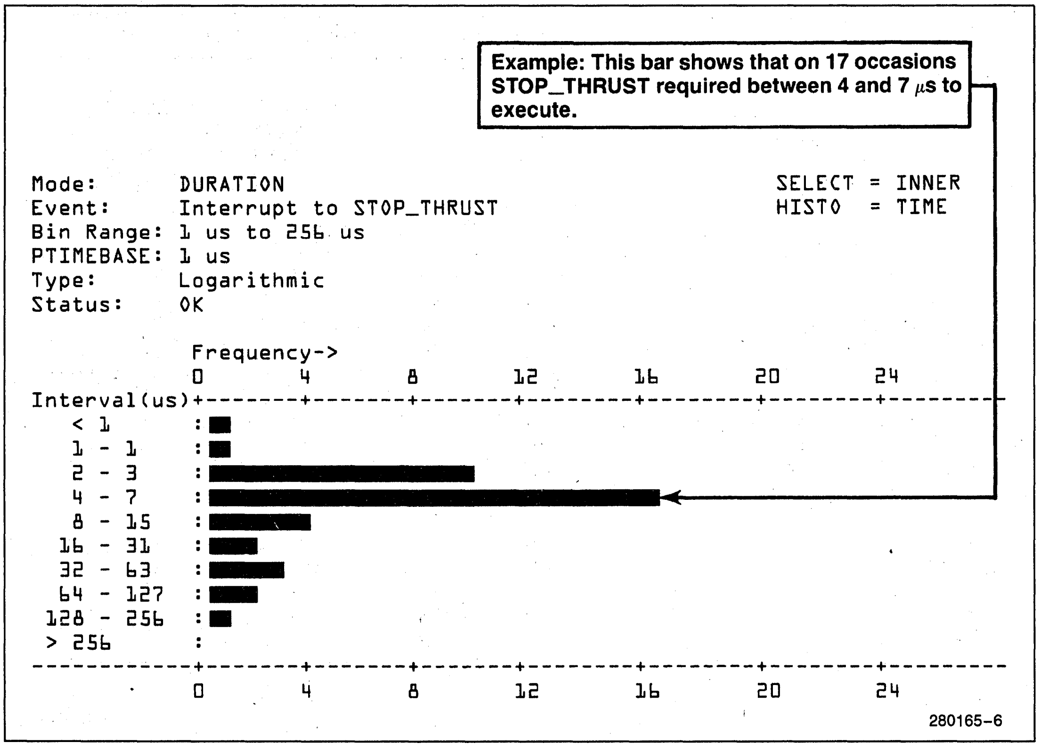
Duration-Mode Analysis: With the duration mode you can focus on timing information for one block of code or one interrupt-address pair. If you wish to determine how regularly a procedure meets performance specifications for timing, duration mode will provide the answer. This mode also is useful when you want information on how widely response time varies between the arrival of an interrupt and the execution of a particular service routine.

Duration mode collects data from repeated executions of a specified block of code or interrupt-address pair. The data is then placed in a number of bins (selectable as 8, 16, or 32 bins). You can select whether the bins have equal intervals or bin size increases logarithmically (use the latter when you expect a wide variation in time values).

Figure 5 shows a sample duration-mode default display. It assumes that a user wishes to find out the variation in response time for a specific interrupt-address sequence. In this case, the user is interested in the elapsed time between an interrupt caused by ground contact of an airplane's landing gear and the execution of the first statement in the procedure that controls thrust shutdown. The display shows, for example, that the bin for the elapsed time interval 4 μs to 7 μs recorded 17 instances of the interrupt-procedure execution pair. Note that in this case the performance specification indicated that elapsed time should never exceed 64 μs, the duration display shows that the current design does not meet the specification.

Duration-Mode Displays: The default duration display (as shown in Figure 5) provides a time histogram. A table display can also be selected.

In duration mode, you are not restricted to learning only about timing that occurs between two events. You can also learn about timing that occurs outside the event pair—the demand for the event pair. Suppose, for instance, RAM memory in your operating system is currently filled, and you want to determine whether one of the processes stored there is used too infrequently to justify its placement in RAM. Collect data on this process using the duration mode. Then use the duration-mode OUTER display option.



Example: This bar shows that on 17 occasions STOP_THRUST required between 4 and 7 μs to execute.

```
Mode:       DURATION                                          SELECT = INNER
Event:      Interrupt to STOP_THRUST                          HISTO  = TIME
Bin Range:  1 us to 256 us
PTIMEBASE:  1 us
Type:       Logarithmic
Status:     OK

             Frequency->
             0       4       8       12       16       20       24
Interval(us)+-------+-------+-------+--------+--------+--------+--------
   < 1      :
   1 - 1    :
   2 - 3    :
   4 - 7    :
   8 - 15   :
  16 - 31   :
  32 - 63   :
  64 - 127  :
 128 - 256  :
   > 256    :
------------+-------+-------+-------+--------+--------+--------+--------
             0       4       8       12       16       20       24
```

280165-6

Figure 5. Duration Mode: Histogram Display

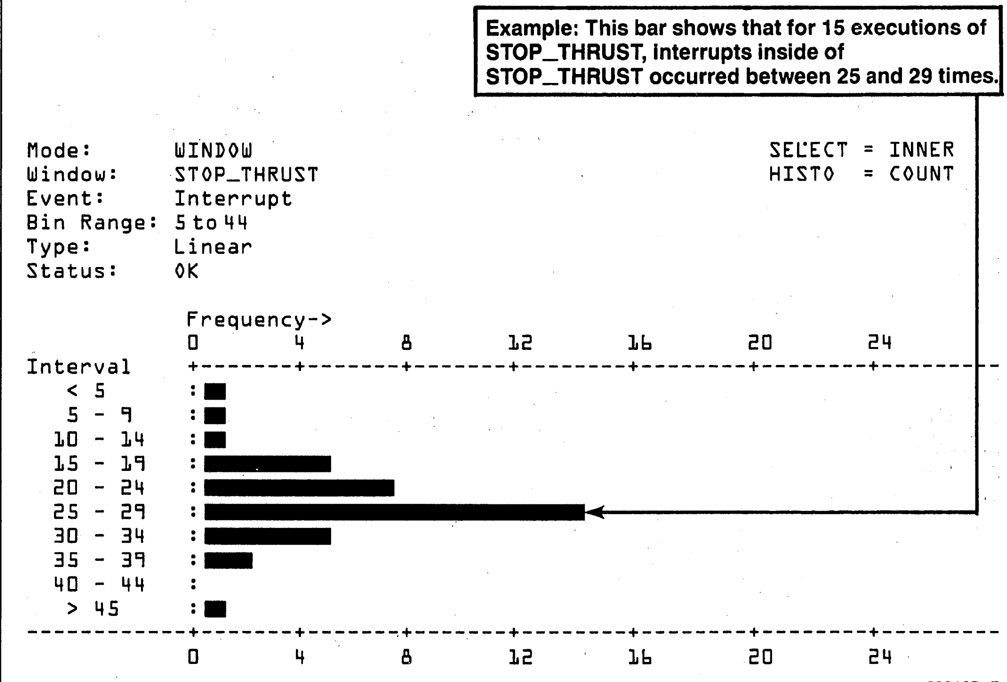
By doing so, you select a display of binned timing data that shows the distribution of the specified process's demand. If the process is infrequently used (contrary to original expectations), it could be moved to disk and RAM space made available for other, more frequently used, routines.

Windowed-Event-Count Mode

The windowed-event-count mode counts how often a specified begin-end pair (window) is entered—and how often, once the window is entered, an interrupt occurs or a specified address is executed. (A count is also kept of how often the selected event occurs outside the window.) As with the duration mode, data is binned. The begin-end pair can be two addresses (specified absolutely or symbolically) or an address and the occurrence of an interrupt.

This mode is useful for obtaining refined count data. For example, if profile mode indicates that procedure A is using excessive time and that much of the time is attributable to procedure calls, you can use this mode to get a better understanding of the situation. Use procedure A as the window and the name of a procedure it calls (B) as the event of interest. Data will then be gathered and placed in bins. The resulting display will show the distribution of how often procedure B is called each time procedure A is executed. Thus, you can see whether procedure B is the procedure causing procedure A to use so much time.

Because the event is counted both inside and outside the window, you can use this mode to determine whether an undesired event occurs excessively within a given block of code. If, for example, one procedure consumes too much time and you suspect that interrupts are occuring excessively during the procedure, use this mode to corroborate your suspicions. Specify the procedure as the window and interrupts as the event. Then display the results both for interrupts within the procedure and those outside the procedure. By comparing the two displays, you can determine whether interrupt frequency within the procedure is skewed. Figure 6 shows a sample display for interrupts that occur inside the window.


Example: This bar shows that for 15 executions of STOP_THRUST, interrupts inside of STOP_THRUST occurred between 25 and 29 times.

```
Mode:       WINDOW                                      SELECT = INNER
Window:     STOP_THRUST                                 HISTO  = COUNT
Event:      Interrupt
Bin Range:  5 to 44
Type:       Linear
Status:     OK

            Frequency->
            0       4       8       12       16       20       24
Interval    +-------+-------+-------+--------+--------+--------+--------
   < 5      :
   5 - 9    :
  10 - 14   :
  15 - 19   :
  20 - 24   :
  25 - 29   :
  30 - 34   :
  35 - 39   :
  40 - 44   :
   > 45     :
------------+-------+-------+-------+--------+--------+--------+--------
            0       4       8       12       16       20       24
```

280165-7

Figure 6. Windowed-Event-Count Mode: Interrupt Latency Histogram

As with the duration mode, you can select the granularity of data collection for the windowed-event-count mode (8, 16, or 32 bins), and you can specify linear or logarithmic binning.

Linkage Mode

Linkage mode has two options, the many-to-one option and the many-to-many option. Both options allow you to focus on inter-procedure activity.

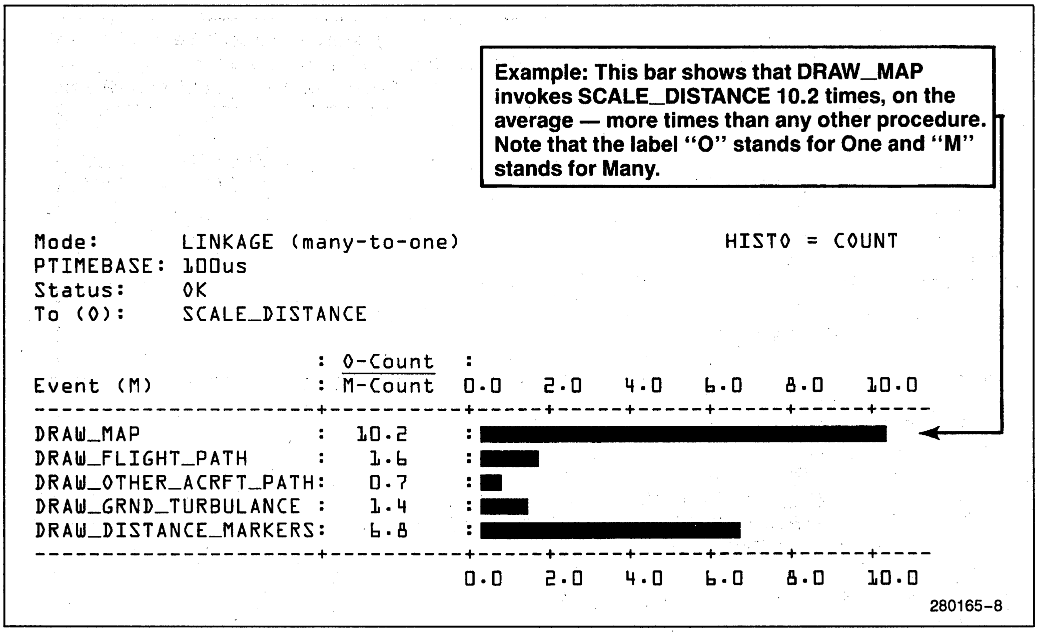
Many-to-One Option: With this linkage option, you can focus on one procedure or block of code (the one) and determine its linkage to other procedures or blocks of code (the many) that call it.

For example, suppose that profile mode has revealed procedure SCALE_DISTANCE to be using excessive time and is called often (see Figure 7). If many of the calls to it are from one or two procedures, to improve execution speed SCALE_DISTANCE could be optimized and moved in line into the procedures that call it often. The many-to-one option can help in this case. You simply enter the PAT LINKAGE analysis command and specify the names of the procedure that call SCALE_DISTANCE (the many) and specify SCALE_DISTANCE as the one. Then, when the program is executed, appropriate count and time data is collected. Figure 7 shows a sample count histogram display for the many-to-one option. For each of the calling procedures, Figure 7 shows the average number of invocations of SCALE_DISTANCE. We see that procedure DRAW_MAP, on the average, invokes SCALE_DISTANCE 10.2 times.

The many-to-one display can also be changed to a time histogram (showing, for each of the many procedures, the average time the one procedure uses) or to a table.

Many-to-Many Option: This linkage option allows you to collect information on the linkage between many event pairs.

In the other modes, you cannot use an interrupt or the same address to specify both members of an event pair. For the many-to-many option, there is no such restriction. Thus, with this option you can collect timing and count information on recursive procedures and interrupt-to-interrupt activity.



Example: This bar shows that DRAW_MAP invokes SCALE_DISTANCE 10.2 times, on the average — more times than any other procedure. Note that the label "O" stands for One and "M" stands for Many.

```
Mode:        LINKAGE (many-to-one)                          HISTO = COUNT
PTIMEBASE:   100us
Status:      OK
To (O):      SCALE_DISTANCE

                          : O-Count   :
Event (M)                 : M-Count  0.0   2.0   4.0   6.0   8.0   10.0
--------------------------+----------+-----+-----+-----+-----+-----+----
DRAW_MAP                  :  10.2    :█████████████████████████████
DRAW_FLIGHT_PATH          :   1.6    :████
DRAW_OTHER_ACRFT_PATH:        0.7    :█
DRAW_GRND_TURBULANCE :        1.4    :███
DRAW_DISTANCE_MARKERS:        6.8    :███████████████████
--------------------------+----------+-----+-----+-----+-----+-----+----
                                     0.0   2.0   4.0   6.0   8.0   10.0
```

280165-8

Figure 7. Linkage Mode (Many-to-One): Count Ratio Histogram

Data for the many-to-many option is displayed in a table. See Figure 8 for a sample display.

USER INTERFACE

The iPAT software is integrated with the emulator software. For example, iPAT command options are integrated in the emulator syntax menu at the bottom of the screen.

In addition, the emulator LITERALLY command can be used to abbreviate frequently used commands. The history buffer is also available to retrieve previous commands.

As already noted, the iPAT analyst requires only one command line to set up an analysis-mode (PAT INIT) and one to request a data display (PAT DISPLAY). There are also six display pseudo-variables used to set display options: SHOW, ABS, SELECT, FILTER, SORT, and HISTO.

Users can save test-coverage data collected for subsequent reviewing. The command PAT SAVE saves coverage data to a user-specified file; the command RECALL enables you to restore the file and then update the test information with additional test runs.

Displays for all modes can be saved to a file using the emulator LIST command.

To speed command entry, you can create registers that save frequently used commands. Then use the names of the desired registers with your analysis and display commands.

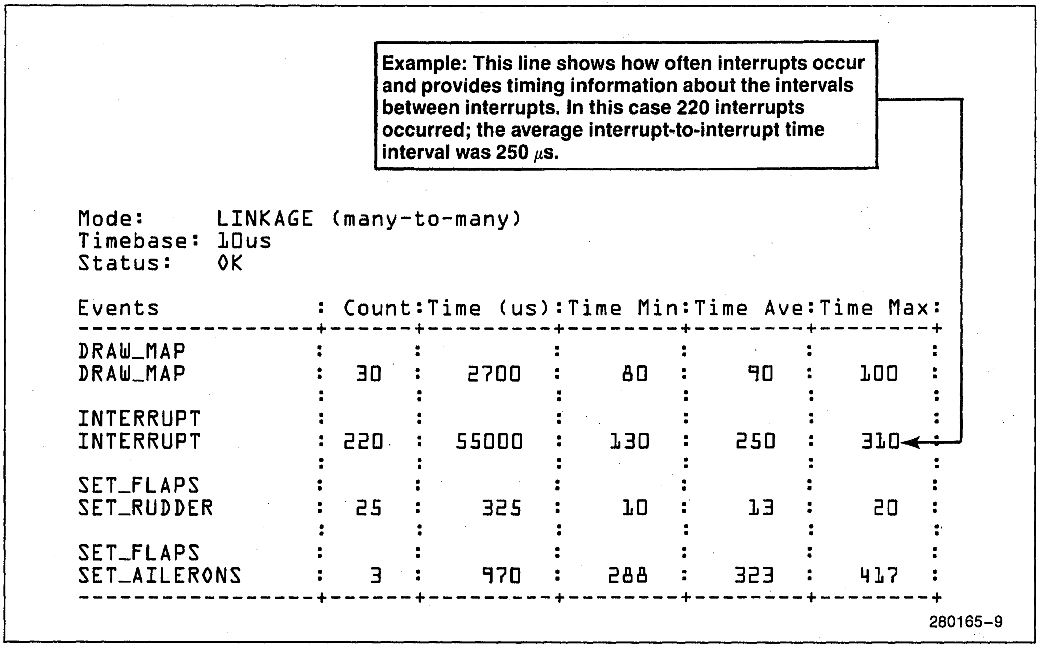
The emulator's screen editor can be used to examine and modify source code that the iPAT analyst has pinpointed as needing improvement.

SPECIFICATIONS FOR iPAT™ AN I²ICE™ SYSTEM

Host Requirements

Intel Series III or Series IV development system ; or an IBM PC XT or PC AT system

At least 512K bytes of RAM (of which 384K bytes must be available for the iPAT/I²ICE system software)



Example: This line shows how often interrupts occur and provides timing information about the intervals between interrupts. In this case 220 interrupts occurred; the average interrupt-to-interrupt time interval was 250 μs.

```
Mode:      LINKAGE (many-to-many)
Timebase:  10us
Status:    OK

Events          : Count:Time (us):Time Min:Time Ave:Time Max:
----------------+------+---------+--------+--------+--------+
DRAW_MAP        :      :         :        :        :        :
DRAW_MAP        :  30  :   2700  :    80  :    90  :   100  :
                :      :         :        :        :        :
INTERRUPT       :      :         :        :        :        :
INTERRUPT       : 220  :  55000  :   130  :   250  :   310<-:
                :      :         :        :        :        :
SET_FLAPS       :      :         :        :        :        :
SET_RUDDER      :  25  :    325  :    10  :    13  :    20  :
                :      :         :        :        :        :
SET_FLAPS       :      :         :        :        :        :
SET_AILERONS    :   3  :    970  :   288  :   323  :   417  :
----------------+------+---------+--------+--------+--------+
```

280165-9

Figure 8. Linkage Mode (Many-to-Many) Display

Available serial channel that operates at 300, 1200, 9600, or 19200 baud. (For a Series IV host, the available channel must be the IEU channel and, to use the iPAT analyst at baud rates greater than 300, an SPU board must be installed.)

Two double-density diskette drives or a hard disk

I²ICE™ System Requirements

Version 1.7 (or greater) probe software

iPAT software does not support I²ICE system operation with the Intel Logic Timing Analyzer (iLTA) and iLTA software does not support iPAT operation.

After the iPAT analyst interface board is installed, space is available in the I²ICE system instrumentation chassis for only one optional board. (Thus the user can install only one optional high-speed (OHS) memory board.)

Only one iPAT analyst will function in a multiple-probe I²ICE system.

iPAT™ Analyst Software

I²ICE host software that includes iPAT software

iPAT confidence tests

iPAT tutorial software

System Performance

Address Range Specification: Address ranges can be specified symbolically (for code compiled by Intel compilers) or with absolute addresses. Addresses anywhere within processor address space can be used.

Speed: The iPAT analyst captures instruction addresses at full processor speeds (however, when users specify many short intervals that are frequently executed, iPAT processing overflow may occur).

Timebase: Data collection timebase selectable as 200 ns, 1 μs, 10 μs, or 100 μs.

Display Updates: Users can specify how frequently (in seconds) displays are updated.

Status: If time-count, bin-count, or FIFO overflow occurs, the display indicates the overflow.

Profile Mode: Collects time and count information on specified entry-exit pairs. Permits specification of 125 entry-exit pairs when calls to other procedures are included in data collection and a minimum of 63 pairs when calls are excluded. Data collection can focus on modules, procedures, lines, absolute address pairs, or interrupt-address pairs. Displays are selectable as histograms or tables; data displayed can be sorted by address, count, or time.

Coverage Mode: Provides up to 252K bytes of coverage, mappable anywhere within the processor address space. Results are displayed in a table; users can select whether the table shows modules, procedures, or lines executed (and/or not executed).

Linkage Mode: The linkage mode has two options:

Many-to-One Option: Collects count and time data about interaction of one specified entry-exit pair with respect to other specified entry-exit pairs. Permits specification of 63 entry-exit address pairs for the many and one entry-exit address pair for the one. Displays are selectable as histograms or tables; data displayed can be sorted by address, count, or time.

Many-to-Many Option: Collects count and time data on one or more pairs of events. Permits specification of 63 event pairs; each member of a pair can be an address or interrupt. Measurements of recursion and interrupt to interrupt are supported. Display is a table.

Modes that Organize Data into Bins: The following two iPAT modes organize collected data into bins. Users can select bin granularity (8, 16, or 32 bins) and the highest and lowest values for the outer bins. Users can also select whether bin intervals are equal or increase logarithmically.

Windowed-Event Count Mode: Collects count data concerning an event that occurs within a specified window. Permits selection of the window entry-exit pair as an address pair, interrupt-address pair, or address-interrupt pair. The event selected can be an address or an interrupt. Resulting binned count data can be displayed as a histogram or table.

Duration Mode: Collects time information for a selected entry-exit pair. Permits selection of an entry-exit pair as an address pair, interrupt-address pair, or address-interrupt pair. Resulting binned timing information can be displayed as a histogram or table.

PHYSICAL CHARACTERISTICS

Target-Interface Board (to be installed in I²ICE system instrumentation chassis):	
Length	30 cm (12 in)
Width	30 cm (12 in)
iPAT Core Module:	
Length	35 cm (13¾ in)
Width	21 cm (8¼ in)
Height	4 cm (1¾ in)
iPAT Power supply:	
Length	28 cm (11 in)
Width	11 cm (4¼ in)
Height	19 cm (7¾ in)

AC power cord for the power supply: 3.0 m (10 ft)

Power-supply-to-core DC power cable: 1.8 m (6 ft), 10 conductor

Emulation-clips jumper cable: 20 cm (8 in), 40 conductor

Execution-trace jumper cable: 10 cm (4 in), 60 conductor

iPAT-to-emulator cable: 0.9 m (36 in), 60 conductor

RS232 serial cable (for connecting the iPAT core to the host system): 3.7 m (12 ft). This cable is shipped with the iPAT software.

Electrical Characteristics

Selectable AC power source: 100V, 120V, 220V, 240V

47–63 Hz

2 amps (AC) at 100V or 120V, 1 amp at 220V or 240V

Environmental Requirements

Operating Temperature: 10°C to 40°C (50° to 104°F)

Operating Humidity: Maximum of 85% relative humidity, non-condensing

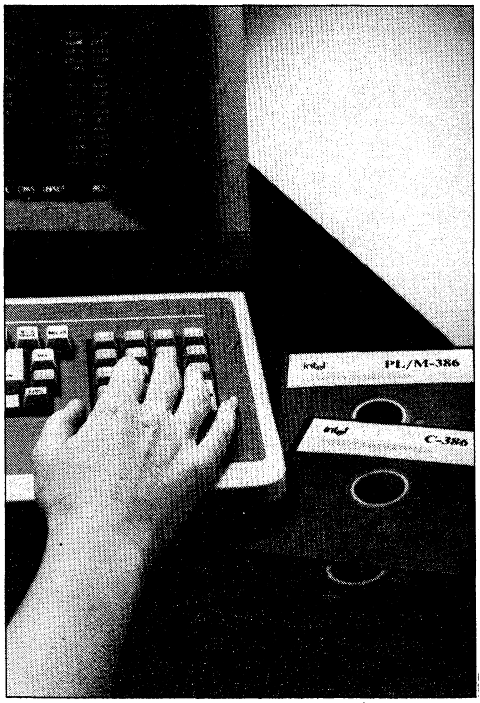
ORDERING INFORMATION

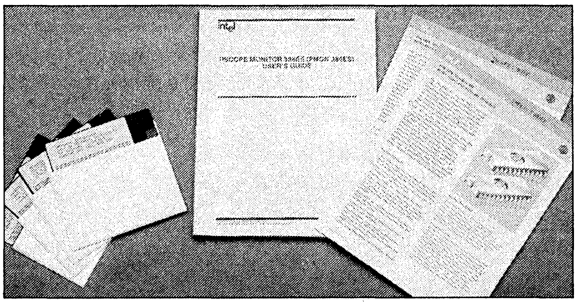
Order Code	Description
iPATCORE	iPAT core unit that supports Intel 8- and 16-bit microprocessors. It must be used with the appropriate emulator target interface, cables, and software.
iPAT86PC	iPAT-I²ICE system target interface, cables, and DOS software for IBM PC AT and PC XT host
iPAT86S3	iPAT-I²ICE system target interface, cables, and ISIS software for Series III host
iPAT86S4	iPAT-I²ICE system target interface, cables, and iNDX software for Series IV host
iPAT86DOS	iPAT DOS software (for use with IBM PC AT and PC XT hosts) and serial cables
iPAT86NDX	iPAT Series IV (iNDX) software and serial cable
iPAT86ISS	iPAT Series III (ISIS) software and serial cable

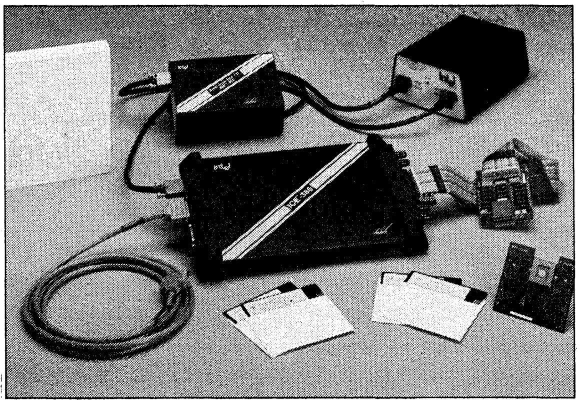
80386 DEVELOPMENT ENVIRONMENT

INTEL HAS THE COMPLETE DEVELOPMENT SOLUTION FOR YOUR 80386 DESIGN:

- **Languages**
  - —Assembler
  - —C
  - —PL/M
- **Debuggers**
  - —PSCOPE Monitor, P-MON 386
  - —ICE™ 386 In-Circuit Emulator
- **Development Hosts**
  - —XENIX* System 310AP
  - —VAX**/VMS system
  - —IBM PC AT or compatible
  - —Intellec Series IV and Compilengine
- **Execution Environment**
  - —iSBC® 386/20 Single-Board Computer
  - —iSBC® 386/100 Single-Board Computer







ORDER NUMBER 280177-002

80386 DESIGN ENVIRONMENT — TO GET STARTED TODAY

You can begin your design based on the 80386 microprocessor today with a powerful set of design tools optimized for the 80386. Intel offers a full set of tools, including languages and debuggers, that let you apply the full power of the advanced, 32-bit 80386 to your product and get your project done quickly and smoothly.

Hosting for the tools is available today on Intel's System 310 running the XENIX* 286 operating system.

ORDERING INFORMATION

Product Description	Order Code
80386 Languages	
—80386 Assembler	X286ASM386PP
—80386 Relocation, Linkage and Library Tools	X286RLL386PP
—PL/M 386 Compiler	X286PLM386PP
—C 386 Compiler	X286C386PP
80386 Debuggers	
—PSCOPE Monitor, P-MON 386ES. Basic 80386 debugging functions including download from host system. Will be updated to pre-production level in late 1986.	X286PMON386ES
—ICE™ 386 In-Circuit Emulator	(Available mid-1986)
Development Host	
—XENIX* System 286/310AP-44. Includes System 310AP, XENIX Operating System, OpenNET networking hardware and software, and four-channel communications controller.	PSYS310AP44XN
Requires 1 MByte memory expansion.	SBC010EX
- or -	
—XENIX* System 286/310AP-46. Includes System 310AP, XENIX Operating System, OpenNET networking hardware and software, twelve-channel communications controller and 2 MBytes memory.	PSYS310AP46XN
Execution Environment	
—iSBC 386/20 Starter Kit ES. Includes iSBC 386/20P Single-Board Computer, 2MB memory board, 386 debug monitor, cables, and documentation.	SBC38620SPKG
—iSBC 386/100ES Kit. Includes iSBC 386/100ES Single-Board Computer, 1MB memory board, 386 debug monitor, cables, and documentation.	SBC386100ES

Hosting Options for 80386 Development Tools

Host	System 310 XENIX*	PC AT DOS	VAX** VMS**	VAX** UNIX†	Series IV	Compilengine
80386 Languages	NOW	2H86	2Q86	1H87	1H87	1H87
PSCOPE Monitor, P-MON 386ES	NOW	—	—	—	—	—
PSCOPE Monitor, P-MON 386	2H86	2H86	—	—	1H87	—
ICE™ 386	2H86	2H86	—	—	1H87	—

*XENIX is a trademark of Microsoft Corporation
**VAX/VMS are trademarks of Digital Equipment Corporation
†UNIX is a trademark of Bell Laboratories

intel®

APPLICATION NOTE

AP-243

November 1986

Debugging with Intel on the VAX*

BRIAN VALENTINE AND STEVE CAPIE
DSO APPLICATIONS ENGINEERING

Order Number: 231479-001

I. INTRODUCTION

The VAX*/VMS mainframe computer has always been popular for large team software development. Its ability to serve many users and the quality software tools available make it a natural choice in the engineering lab. The introduction of quality microprocessor development languages, such as the 8086, 80186 and 80286 series from Intel Corporation, has increased the versatility of the VAX, further strengthening its position as the large team software development standard machine.

However, microprocessor development is not solely software development. At some time, the VAX developed software must be integrated and debugged in the target microprocessor environment. There are no suitable VAX tools for this integration/debug phase, so the developer must turn to a development system supplied by the microprocessor manufacturer. Intel, for example, provides a Series IV development system, coupled with a sophisticated in-circuit emulator and integrated instrumentation system, to solve this difficult integration/debug phase. Consequently, software developers must use both a VAX computer and microcomputer development system (MDS) workstation to perform their microprocessor development.

Currently VAX software developers need two keyboards to perform their work efficiently:a VAX keyboard for software generation and a microcomputer development system workstation for debugging and in-circuit emulation. Often these two keyboards are not located in the same working environment. While the microcomputer development system workstation is generally placed in an engineering laboratory, the VAX keyboard is typically found in the software developer's office. Having to work in both environments creates two problems for the software developers. First, debugging or using an in-circuit emulator requires him to leave his office, second, he's required to learn two keyboard environments.

One solution to both these problems is to allow software developers to perform debugging and in-circuit emulation from their offices using their VAX keyboard. With Intel's new VAX program called CONNECT, this is now possible. CONNECT allows software developers to use their VAX keyboards as a virtual MDS workstation. Software developers can now run PSCOPE, ICE TM and I2ICE TM debuggers from their VAX keyboard.

Some of the highlights of CONNECT include:

1. A user friendly human interface
   — help command
   — informative and easy to understand system messages
2. A flexible menu for selecting MDS workstation.
3. The support of a VAX environment with multiple VAX/MDS workstations (performs keyboard lock out).
4. Source code is supplied with package; therefore, CONNECT may be customized to your needs.

With CONNECT, software developers can now do the following:

1. Spend less time walking from keyboard to keyboard, and spend more time developing software.
2. By only using their fingertips, access any MDS workstation connected to the VAX.
3. Perform the entire development process in the comfort of their own office.

CONNECT is another innovative feature to add to the Intel-VAX microprocessor development environment. Intel already provides cross software which includes compilers, linkers and locators. In addition, for file transfer between the VAX environment and the MDS environment, Intel provides a serial communication link (ACL) and an Ethernet communication link (NVL).

The remainder of this application note will cover the following material:

II. CONNECT environment
   A. Hardware considerations
   B. Software considerations
III. CONNECT SETUP - for the VAX system manager
   A. How to create CONNECT:CONNECT.dat configuration file
IV. USING CONNECT—for the VAX software developer
   A. How to run program CONNECT with qualifiers
   B. How to choose and select MDS workstation
   C. How to use VAX keyboard as a virtual MDS workstation
   D. Additional CONNECT commands

II. CONNECT ENVIRONMENT

A. Hardware considerations

The VAX system manager must connect all desired MDS workstations to the VAX, using RS-232 serial cables. These cables are connected from any VAX serial port to serial channel 1 on the MDS.

MDS is used in this applications note as an abbreviation for Microcomputer Development System
MDSTM a trademark of Mohawk Data Sciences
*VAX is a trademark of Digital Equipment Corporation

Example: Connect an RS-232 cable from port TTA1:on the VAX to serial channel 1 on the MDS Series IV development system.

Each MDS to be used with the CONNECT requires a serial cable from serial channel 1 to a port on the VAX. The RS-232 connection is defined as:

Pins		Pins
1	⟷	1
2	⟷	2
3	⟷	3
4	⟷	4
5	⟷	5
6	⟷	6
7	⟷	7
20	⟷	20

NOTE:
If there are problems with CONNECT communicating to the MDS, it is likely a serial cable problem. Depending on the jumper configuration of the MDS serial channel 1, a different configuration of the serial cable may be needed. The serial cable above will work with a factory standard jumper configuration of serial channel 1 on the MDS. Consult the Intellec Series II Microcomputer Development System Hardware Reference Manual #980556-002 for more information on the serial channel 1 configurations for a Series II, III or IV.

B. Software considerations

Now that a serial cable is installed linking the MDS to the VAX, all normal console input and output of the MDS should be redirected to serial channel 1.

To redirect console input and output of a Series II or Series III to serial channel 1 perform the following steps:

1. With the ISIS prompt (-) on the screen type:

```
-SPEED 300 | 600 | 1200 | 2400 | 4800 | 9600
-CONSOL :TI:,:TO:
```

SPEED is used to set the baudrate of serial channel 1 while CONSOL is used to redirect console input and console output to serial channel 1 on the Series II or III. Running CONNECT, while if there is a need to restore the console input and output back to the MDS screen, enter the following command:

```
-CONSOL :CI:,:CO:
```

NOTE:
If at any time the Series II or III encounters an error that requires a warm boot (Example:ERROR 30), the console will be automatically switched back to the normal console input and output. Consequently, any time a warm boot occurs, the console input and output must be redirected to serial channel 1.

To redirect console input and output of a Series IV to serial channel 1, run the STTY program with the baudrate no greater than 2400 baud. See the Intellec Series IV Operating and Programming Guide #121753-004 for more detail on the STTY program. Example SIV/STTY command:

```
>STTY BAUDRATE (2400)
CONFIG(VT100.CFG) REMOTE
```

When running CONNECT, if there is a need to restore the console input and output on the Series IV, enter the following command:

```
>STTY LOCAL
```

Please note that while the Series II or Series III is running CONSOL and the Series IV is running STTY, no user may work at these workstations.

NOTE:
All VAX serial ports that are used to connect the MDS to the VAX should be configured at a constant baud rate. The autobaud feature should be disabled. The suggested baudrate is 2400 baud for Series IV and 9600 baud for Series II.

III. CONNECT SETUP - FOR THE VAX* SYSTEM MANAGER

A. How to create CONNECT:CONNECT.DAT configuration file

(Only a System Manager needs to read this)

NOTE:
CONNECT requires a system logical name to be assigned to the directory on the VAX where the CONNECT image and configuration files will be stored. The logical name must be "CONNECT:". Also, see APPENDIX A for a listing of the CONNECT. CLD command definition file. This file should be added to the system-wide DCL command file. If this is done, any VAX user may access and execute CONNECT.

A system manager must create an ASCII file named "CONNECT:CONNECT.DAT". This file contains all the configuration information of the MDS workstations connected to the VAX. This file is used by connect to determine what type of MDS systems have been connected to the VAX. eg. Series II, III, etc. One of the features of CONNECT is that a user may select different workstations to use during debugging; CONNECT uses the configuration file to determine which port on the VAX the corresponding workstation is connected.

The file consists of the following configuration parameters: system, id, ice, port.

"System" refers to the type of MDS workstation. eg. Series 2/3/4.

"Id" refers to the identification number of the workstation (system manager chooses this). This ID number is any number from 00 to 99. Each system entered in the configuration file must have a unique ID number. It is suggested that when building the configuration file, the first system entered is given id = 00, the second system id = 01, etc.

"Ice" refers to the type of ICE module connected to the MDS.

"Port" refers to the serial address to which the workstation is connected to the VAX (e.g. TTA0:). For example, if you connected a Series IV to the port TTA7:on the VAX, you would enter port=tta7:in the configuration file.

NOTE:
The maximum line length for a line in the CONNECT.DATconfiguration file is 128 characters.

These four configuration parameters accompanied by their input values form a single MDS workstation record.

Each parameter and its input value are specified on a SINGLE LINE of input in the file. Each record must contain both the **system** and **port** configuration parameters. Both id and ice are optional. THE **SYSTEM** PARAMETER MUST ALWAYS OCCUR FIRST IN EACH RECORD. to add comments in the file, enter a semicolon in the first position of a new line. The syntax for each parameter and it's input value is as follows:

The system manager should now perform the following steps:

1. Go into a VAX editor (e.g. EDT) and create a new file called CONNECT:CONNECT.DAT
2. Follow the rules in forming the MDS workstation records. If there is a Series III with an ICE-51 connected to the VAX and a Series IV with an ICE-86, the configuration file created could look as follows (the semicolons are in column zero):

```
; This series 3 is in SJC's office
; Has a 557 board (224K addition)
system : 3
id : 00
ice : 51
port : ttb0
; This series IV is in LAB 2
; Note the next record uses the
; abbreviations for the parameters
s : 4
id : 01
ic : 86
p : ttb1
```

3. After the configuration file has been created, exit the editor. Now it is time to check the data entered to be sure it is in a form CONNECT can understand. To do this, run the program VCONFIG supplied with the CONNECT package. Example: RUN CONNECT:VCONFIG
4. If any errors were found, VCONFIG will print the line number, the line with the error, and a message explaining what error occurred. Fix the error by re-editing the CONNECT:CONNECT.DAT and re-run VCONFIG.
5. If no errors were detected, then VCONFIG will report

   NO ERRORS DETECTED IN ASCII FILE. SYSTEM SUCCESSFULLY CONFIGURED!
6. Be sure to set world read protection on the CONNECT:CONNECT.DAT file.

```
S(ystem)  : [ 2 | two    |   ii ]                          →  REQUIRED  ←
            [ 3 | three| iii ]
            [ 4 | four|   iv]

ID        : [ 0..9 ] [ 0..9 ]                              →  optional  ←

IC(e)     : [ 42 | 44 | 49 | 51 | 85 | 86 | 88 ]           →  optional  ←
            [ i2ice ] - [ 86 | 88 | 186 | 188 | 286 ]

P(ort)    : tt[ a..z ] [ 0..9 ]                            →  REQUIRED  ←
```

NOTE:
IF the MDS configuration cannot be specified with the given parameters, it is suggested that the user assign the SYSTEM, ID and PORT parameters. Therefore, when a development engineer connects to the system with CONNECT, the ID number may be specified. Also, if the system is not a II, III, or IV, any number type may be assigned to the system (e.g. II, II or IV), but when the system is connected to with CONNECT, the ID number must be used instead of the system type.

IV. USING CONNECT - FOR THE VAX SOFTWARE DEVELOPER

A. How to run the program CONNECT

NOTE:
The CONNECT.CLD file must be installed on the VAX before CONNECT will run.

With the dollar sign ('$') prompt present, type in the following format to run CONNECT:

```
$CONNECT [/baudrate = (SPEED)]
[/macrofile = (FILENAME)] [/[NO]VT100]
```

SPEED is one of the following: 300, 600, 1200, 2400, 4800, 9600

FILENAME is any legal filename on the VAX

If the baudrate qualifier is omitted, then the default is set to 2400 baud. If the NOVT100 qualifier is omitted, then the default is set to VT100. The macrofile qualifier is explained later.

B. How to choose and select workstations when running CONNECT

When running the program CONNECT, the following header will appear:

```
CONNECT    Version (X$$$)
```

At this point CONNECT checks the validity of the CONNECT:CONNECT.DAT file. This is done by running the VCONFIG program. If an error was found, CONNECT will display to the user:

```
AN ERROR WAS FOUND IN THE FILE
CONNECT:CONNECT.DAT PLEASE REPORT THIS
TO THE SYSTEM MANAGER
```

At this point, the system manager must find and correct any error(s) in the file CONNECT:-CONNECT.DAT. Otherwise, the same error message will be printed to the screen each time CONNECT is run.

If the user typed in the macrofile qualifier, then the development system search parameters are read from the file entered for the qualifier. Otherwise, the user is prompted with the CONNECT menu screen. The screen is as follows:

```
          --------------------> CONNECT MENU <--------------------

The CONNECT Options are the following:
     E(xit)
     H(elp)
     S(ystem)=attribute, IC(e)=attribute, ID=attribute

To select MDS workstation, enter any combination of the following MDS search
parameters:

Please enter MDS Option:

          The MDS options available to the user are the following:

               S(ystem) =    attribute Distinguishes among a Series 2,3 or 4
               ID       = attribute    Identifies a workstation with an id
number
               IC(e)    = attribute    Chooses between an ICE and I2ICE
workstation

          The attributes for each option are as follows:

               S(system)=[2|two|ii]
                        =[3|three|iii]
                        =[4|four|iv]
                        =[8bit] (Either a Series 2 or Series 3)
                        =[16bit] (Either a Series 3 or Series 4)

               ID       =[0..9][0..9]

               IC(e)    =[42|44|49|51|85|86|88]
                        =[i2ice]-[86|88|186|188|286]
```

None, one, two or any possible combinations of these three options may be used. When two or more of these options are used, THE COMMA IS REQUIRED AS THE DELIMITER. Any option which is omitted will be treated as a wildcard!

```
<EXAMPLES OF MDS OPTION INPUT FROM THE
CONSOLE INPUT>
```

1. If the user wants to connect to any available MDS workstation with an ICE-86 workstation:

   Please enter MDS Option:ic=86

2. If the user wants to connect to the Series II with an ICE-51 workstation:

   Please enter MDS Option:s=ii, ice=51

3. If the user wants to connect to any 8-bit workstation.

   Please enter MDS Option:s=8bit

4. If the user wants to connect to a Series IV with an I2ICE-186.

   Please enter MDS Option:s=four, ic=i2ice-186

5. If the user wants to connect to any available MDS workstation.

   Please enter MDS Option:(cr)

6. If the user wants to use a particular system, to continue a previous debug session, he should specify the station ID number:

   Please enter the MDS option:id=10

Notice that whether the search parameters are read from the user interactively or from the macrofile, there should be NO LEADING, TRAILING, OR ANY SPACES WHAT-SO-EVER ON THE MDS OPTION LINE (NO EXCEPTIONS!!!).

If the input is expected from the macrofile (specified with the MACROFILE qualifier when running CONNECT), CONNECT ONLY LOOKS AT THE VERY FIRST LINE IN THE FILE. The first line MUST END WITH A CARRIAGE RETURN. If the carriage return is not included CONNECT will be unable to properly interpret the line and will print an error to the screen. If the first line in the macrofile only contains the End Of File (EOF) marker, CONNECT will display the error:

```
ERROR IN GETTING LINE FROM
(FILENAME)
<FILENAME> ONLY CONTAINS THE EOF
MARKER
EITHER PUT THE MDS SEARCH
PARAMETER(S) INTO THE FILE OR DELETE
IT ENTIRELY!
<<CONNECT TERMINATED>>
```

```
<EXAMPLE OF MDS OPTION INPUT FROM
MACROFILE>
```

```
ic=i2ice-86,s=four(cr)
[EOF]
```

The previous example would be used if the user wants a series IV with an i2ice-86 workstation.

At this point, CONNECT interprets the line (from either console input or from the macrofile). If any errors were found, CONNECT will show the error, show the user what he/she typed, and give him/her the correct syntax of the incorrect MDS search parameter. If there were no errors, then one of the five following conditions (and possibly displayed messages) will occur:

1. The MDS options could not be met. The message is:

   ```
   NO WORKSTATION WAS FOUND THAT MET YOUR
   MDS OPTION REQUIREMENT(S).
   ```

   This means that the system specified by the user was not found in the CONNECT.DAT file. The system manager must be notified to add the system to the configuration file.

2. The particular device chosen (i.e. id number) is currently being used. The message is

   ```
   PARTICULAR DEVICE IS ALREADY IN USE,
   PLEASE PICK ANOTHER.
   ```

   This indicates that some other user is currently using the system specified.

3. If the first workstation found is busy, CONNECT continues its search. If no other workstation is found, then the message is:

   ```
   ALL DEVICES THAT MEET YOUR MDS OPTION RE-
   QUIREMENT(S) ARE CURRENTLY BEING USED.
   ```

4. A workstation was found, but its port address does not exist on the VAX system. The message is

   ```
   THE SPECIFIED DEVICE (TT**) DOES NOT EX-
   IST IN THE HOST SYSTEM. PLEASE TELL SYS-
   TEM MANAGER OF THE ERROR IN
   CONNECT:CONNECT.DAT ((CONNECT TERMI-
   NATED))
   ```

This last message is fatal and will therefore exit CONNECT. All other messages will re-prompt the user with the MDS option line.

5. A workstation was found and a connection can be made!!!

   The MDS workstation is now ready for use.

CONNECT will display to the screen the MDS characteristics of your workstation and prompt you for verification.

```
        --------MDS Workstation
        Characteristics--------

System is a Series      (ALWAYS APPEARS)
Identification number is ______
ICE workstation is _________________

ARE THESE CHARACTERISTICS ACCEPTABLE
(Y/N):
```

In response to the question on the screen, pressing any key other than 'Y' will re-display the entire CONNECT menu screen. Otherwise, you will be connected via serial channel 1 to the debug workstation. If your screen remains blank, do not worry. If no data is transferred back to your terminal from the workstation within five seconds, the following error message will be displayed:

```
TIME OUT ERROR: THOUGH THE PORT ADDRESS
                EXISTS <TT**>, THERE IS
                NO MDS WORKSTATION CON-
                NECTED TO THE VAX SYSTEM.

NOTE: IF YOU KNOW FOR A FACT THAT THERE IS A
      MDS WORKSTATION CONNECTED TO THAT
      PORT THEN EITHER THE SERIES IV IS NOT
      FUNCTIONING IN STTY PROGRAM
                      OR
      SERIES II/III IS NO LONGER IN CONSOL
      MODE

PLEASE REPORT THIS TO SYSTEM
MANAGER!!!

<<CONNECT TERMINATED>>
```

Otherwise, the Series II/III prompt or the Series IV prompt and menu line will be displayed.

C. How to use VAX keyboard as a virtual MDS workstation

Now that you are connected to the MDS workstation, any key you press or command you type will be interpreted by that workstation. Therefore, your VAX terminal now looks and acts like the console of the MDS.

D. Additional CONNECT commands

The following are additional CONNECT commands which are supported while the VAX keyboard is acting as a virtual MDS workstation:

vbreak or control-b	Breaks connection with MDS workstation and prompts you with the CONNECT menu
vdev	Tells you the system and, if possible, the ID number and ICE workstation
vhelp	Explains all the CONNECT commands listed here
vquit or control-p	Exits the program CONNECT

APPENDIX A
Source code modules for CONNECT

CONNECT is written using the VAX/VMS C Compiler. It is included in the Network/Series IV toolbox or can be purchased from Intel's Insite Users Program Library.

Insite Part Number: CONNECT, Part #AD26

Insite Mailing Address:
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051
Attn:Insite User's Program Library GR1-2-78
Telephone:408-987-7256

CONNECT C source code modules:

CONNECT.C	—main module for CONNECT
VSUBMIT.C	—reads macro file
CONFIGCH.C	— checks errors of CONNECT.DAT configuration file
CCONNECT.C	— searches the configuration file for user's MDS
INTERACT.C	— handles the communications of VAX to MDS
STRUTIL.C	—string parsing routines

VCONFIG C source modules:

VCONFIG.C	— checks for errors in the CONNECT.DAT configuration file
STRUTIL.C	—string parsing routines

Executables:
CONNECT.EXE
VCONFIG.EXE

CONNECT command definition file:
CONNECT.CLD

Command files to generate CONNECT:

V.COM	—generates CONNECT.EXE
VCONFIG.COM	—generates VCONFIG.EXE

Sample CONNECT configuration file:
CONNECT.DAT

Sample CONNECT macro file:
CONNECT.MAC

Series IV Keys Defined on VAX Terminal

The following shows you how to simulate the function keys and toggle key on the Series IV while running CONNECT on your VAX terminal:

Control sequence on VAX terminal	Series IV equivalent key
control-f (unshifted) 0	F0 (unshifted)
control-f (unshifted) 1	F1 (unshifted)
control-f (unshifted) 2	F2 (unshifted)
control-f (unshifted) 3	F3 (unshifted)
control-f (unshifted) 4	F4 (unshifted)
control-f (unshifted) 5	F5 (unshifted)
control-f (unshifted) 6	F6 (unshifted)
control-f (unshifted) 7	F7 (unshifted)
control-f (shifted) 0	F0 (shifted) ON LINE HELP
control-f (shifted) 1	F1 (shifted) ON LINE HELP
control-f (shifted) 2	F2 (shifted) ON LINE HELP
control-f (shifted) 3	F3 (shifted) ON LINE HELP
control-f (shifted) 4	F4 (shifted) ON LINE HELP
control-f (shifted) 5	F5 (shifted) ON LINE HELP
control-f (shifted) 6	F6 (shifted) ON LINE HELP
control-f (shifted) 7	F7 (shifted) ON LINE HELP
control-f t(oggle)	toggle switch

APPENDIX B

```
\* Intel Corporation 1984
   VT1ØØ configuration file

   This file contains the VT1ØØ configuration information for the program
   STTY *\

\* Default configuration for NRM console or Series IV "secondary console"
   (serial channel 1) : *\

AV=25;             \* NUMBER OF LINE (DECIMAL) *\
AW=F;              \* WRAPPER                  *\
AX=F;              \* X FIRST                  *\
AI=T;              \* INVISIBLE ATTRIBUTES     *\
AC=T;              \* CHARACTER ATTRIBUTES     *\
AO=2Ø;             \* OFFSET FOR CURSOR ADDR   *\
AFBK=2Ø;           \* BLANK CHARACTER          *\
AFIG=;             \* IGNORE CHARACTER         *\
AFKM=FF;           \* KEY MASK                 *\
AFPM=FF;           \* PRINT MASK               *\
AFDC=T;            \* DC1 - DC3 PROTOCOL       *\
AFHG=T;            \* HANG UP MODEM ON SW LOCAL *\
AFLO=F;            \* LOGOFF ON CONSOLE SWITCH  *\
AFSE=T;            \* SLOW LINE EDIT MODE       *\
AFTX=Ø4;           \* TRANSMIT READY MASK VALUE *\

\* The following commands are used to convert keyboard sequences on your
   terminal to Series IV codes. *\

AFCU=1B41;                   \* CURSOR UP                   *\
AFCD=1B42;                   \* CURSOR DOWN                 *\
AFCR=1B43;                   \* CURSOR RIGHT                *\
AFCL=1B44;                   \* CURSOR LEFT                 *\
AFCH=ØF;                     \* CURSOR HOME                 *\
AFXZ=1B5B324B;               \* DELETE LINE                 *\
AFXA=1B5B4B;                 \* DELETE RIGHT                *\
AFXU=15;                     \* UNDO                        *\
AR=7F;                       \* RUBOUT                      *\
AFSS=13;                     \* SCREEN SUSPEND              *\
AFSR=11;                     \* SCREEN RESUME               *\
AFCA=Ø3;                     \* COMMAND ABORT               *\
AFDB=Ø4;                     \* DEBUG                       *\
AFJA=1B4F53;                 \* JOB ABORT                   *\
AB=1B;                       \* ESCAPE                      *\
AFSO=;                       \* SCREEN ON/OFF TOGGLE        *\
AFC1=;                       \* CLEAR SCROLL PART OF SCREEN *\
AFC2=;                       \* CLEAR MESSAGE PART OF SCREEN *\
AFC3=;                       \* CLEAR PROMPT PART OF SCREEN *\
AFCC=;                       \* CAUSE INTERRUPT 3 (NMI)     *\
```

231479–1

```
\* The following commands are output sequences which convert screen control
   character sequences from the Series IV sequence to the sequence expected
   by your terminal *\


AFMU=1B41;                 \* CURSOR UP *\
AFMD=1B42;                 \* CURSOR DOWN *\
AFMR=1B43;                 \* CURSOR RIGHT *\
AFML=1B44;                 \* CURSOR LEFT *\
AFMH=1B48;                 \* CURSOR HOME *\
AFMB=ØD;                   \* RETURN (LF) *\
AFER=1B5B4A;               \* ERASE FROM CURRENT POSITION TO END OF SCREEN *\
AFEL=1B5B4B;               \* ERASE FROM CURRENT POSITION TO END OF LINE *\
AFAC=1B2Ø;                 \* CURSOR CONTROL LEAD IN -- CONNECT EXPECTS 2ØH *\

AFAT=;                     \* THERE WILL BE NO INVERSE VIDEO ON ANY OF THE *\
AFRV=;                     \* VAX TERMINALS.                               *\
AFNV=;

AFIL=;                     \* INSERT LINE *\
AFDL=;                     \* DELETE LINE *\

\* ------------------------------------------------------------------------- *\
\* THE FOLLOWING UNSHIFTED/SHIFTED FUNCTION CODES AND TOGGLE SWITCH "MUST"
   REMAIN ON ALL VAX TERMINALS.  ONCE AGAIN, DO NOT CHANGE ANY OF THESE CODES *\

AFLØ=8Ø;                         \* UNSHIFTED FUNCTION KEY Ø *\
AFL1=81;                         \* UNSHIFTED FUNCTION KEY 1 *\
AFL2=82;                         \* UNSHIFTED FUNCTION KEY 2 *\
AFL3=83;                         \* UNSHIFTED FUNCTION KEY 3 *\
AFL4=84;                         \* UNSHIFTED FUNCTION KEY 4 *\
AFL5=85;                         \* UNSHIFTED FUNCTION KEY 5 *\
AFL6=86;                         \* UNSHIFTED FUNCTION KEY 6 *\
AFL7=87;                         \* UNSHIFTED FUNCTION KEY 7 *\
AFUØ=9Ø;                         \* SHIFTED FUNCTION KEY Ø *\
AFU1=91;                         \* SHIFTED FUNCTION KEY 1 *\
AFU2=92;                         \* SHIFTED FUNCTION KEY 2 *\
AFU3=93;                         \* SHIFTED FUNCTION KEY 3 *\
AFU4=94;                         \* SHIFTED FUNCTION KEY 4 *\
AFU5=95;                         \* SHIFTED FUNCTION KEY 5 *\
AFU6=96;                         \* SHIFTED FUNCTION KEY 6 *\
AFU7=97;                         \* SHIFTED FUNCTION KEY 7 *\
AFTS=8A;                         \* TOGGLE SCREEN          *\

\* END OF COMMANDS WHICH CANNOT BE CHANGED FOR ANY VAX TERMINAL *\
\* ------------------------------------------------------------------------- *\
```

231479-2

intel®

APPLICATION NOTE

AP-253

September 1985

Adding Value to Intel's NDS-II Development System Network with CP/M-80

BRIAN VALENTINE
DSO APPLICATIONS ENGINEERING

Order Number: 231533-001

INTRODUCTION

Word processing has long been a desired, if not necessary, feature of a microprocessor development system. Although most systems can support high-level languages, in-circuit emulators, and many other tools to develop microprocessor software and hardware, few support advanced word processing.

A typical microprocessor design project consists of three general phases: design and staffing, implementation, and test and integration. Word processing plays a major part in all three phases. Each phase requires an advanced editor, such as Wordstar by Micropro, to generate the supporting documentation that accompanies a project.

Microcomputer Development Systems for the design engineers should have the capability to access word processing tools. This feature would eliminate the need for a second system or terminal on the engineer's desk. The engineer also should have the ability to share his or her word processing database with other engineers involved in the project.

The design and staffing phase also requires electronic spreadsheets, such as Multiplan by Microsoft, to track potential problems or determine the number of engineers required for the project. The typical engineering support staff are all running upon various different systems. This application note will show how added capabilities to Intel's Network Development System (NDS-II) will provide each engineers workstation with word-processing and spreadsheet capabilities. In addition, since all workstations are connected via the NDS-II and Ethernet, each engineer will have access to other engineers' word-processing databases.

The NDS-II Network

Intel's NDS-II enables development system mainframes to be connected into a network using Ethernet. Additionally, each mainframe can host several ISIS clusters that use low-cost serial lines to support terminals. The complete product line is described in the NDS-II System Description (See Appendix C for complete details).

Low-cost terminals allow everyone to share and manipulate files and data directly on the network. This feature eliminates many intermediate steps required in producing a final document. The addition of CP/M-80 for the NDS-II, combined with word processing and spreadsheet programs, further increases word-processing efficiency. Figure 1 shows a typical NDS-II environment.

SOLUTION

The problem is to add word-processing capability to the NDS-II network. Since CP/M-80 already runs on an Intel Series-II development station in standalone mode, the solution is to modify CP/M-80 to access remote files located on the NDS-II file server instead of local floppies. This solution will provide networking to CP/M-80 and also increase the I/O performance, since all files are now accessed over Ethernet, rather than from slower floppy disk drives.

The method chosen to modify CP/M-80 is to modify the BIOS of CP/M-80. (For more information, see "CPM Alteration Guide," (c) 1979 Digital Research and Appendix C.)

OVERVIEW OF CP/M-80

CP/M-80 contains five sections:

- BIOS – basic I/O system
- BDOS – basic disk operating system
- CCP – console command processor
- TPA – transient program area
- SPA – system parameter area

Figure 2 shows where each of the sections reside in memory on a Intel Series II station. Each section executes specific functions. The BIOS section is responsible for the I/O options to all the system-specific peripherals. When CP/M-80 is ported from one system to another, the BIOS is the only section that changes (length: variable).

The BDOS section is responsible for the general non system-specific operation of the peripherals. BDOS functions call BIOS functions to operate the peripherals. The BIOS and BDOS are collectively know as the full disk operating system (FDOS) (length: fixed).

The CCP section reads commands from the user. It has the following built-in commands: DIR, ERA, REN, SAVE, TYPE and USER (length: fixed).

The TPA section is used as user program space. Application programs are loaded and operate in this section (length: available memory – size of BIOS, BDOS and SPA). Note: If needed, the TPA can occupy space up to the beginning of the BDOS. If this happens, CCP must be reloaded when the program running in the TPA exits.

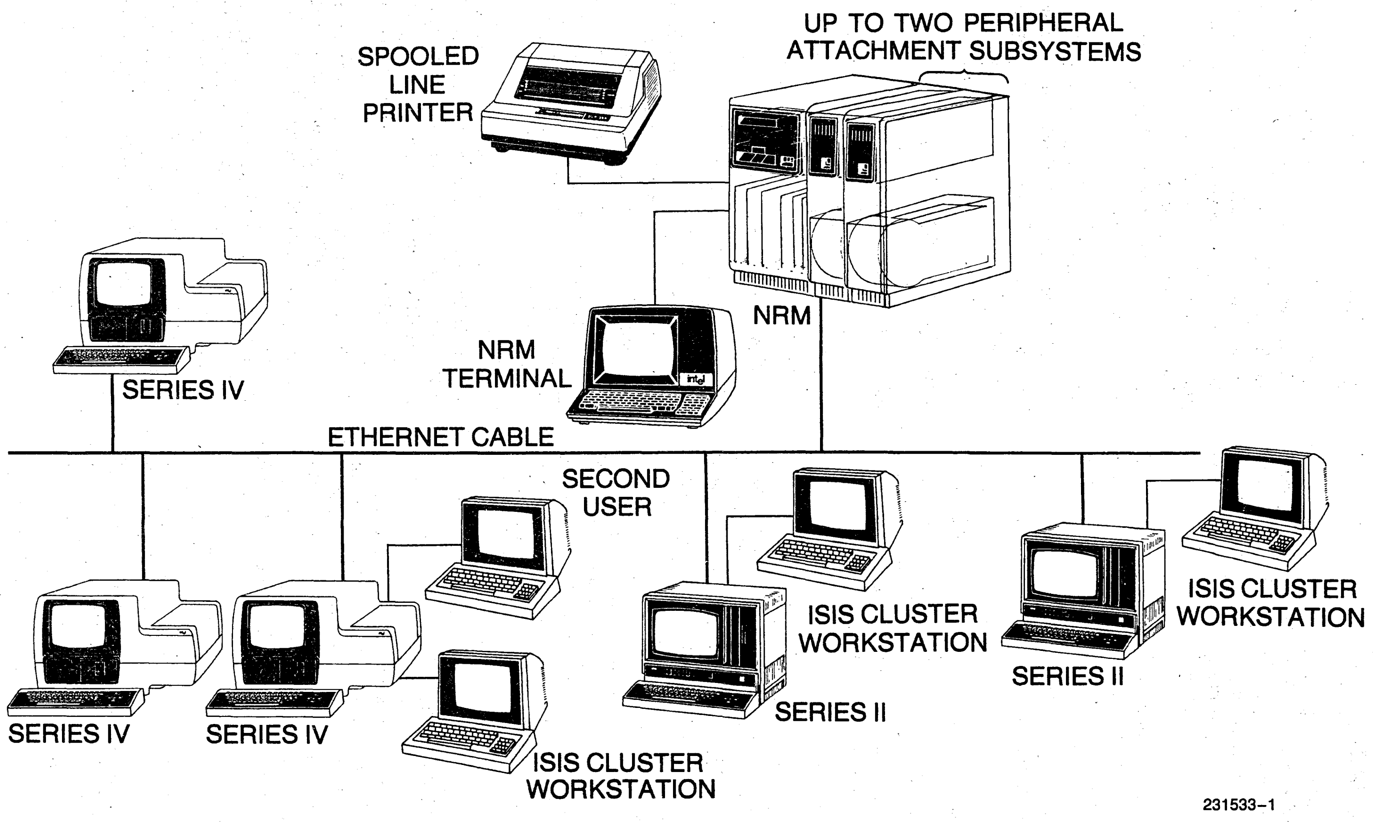


Figure 1. NDS-II Configuration

The SPA section contains operating system information such as the current disk, user number, peripheral assignments, the start address of the BIOS and BDOS sections, restart locations and the default buffer. Figure 3 shows a breakdown of the SPA (length: 256 bytes).

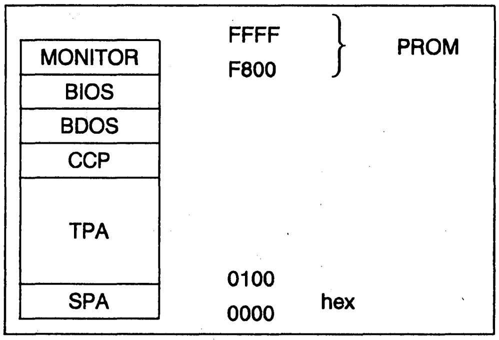


Figure 2. Series II Memory Map of CP/M-80

BIOS FUNCTIONS

Now that CP/M-80 has been broken down, the problem is how to port the Series II standalone version of the BIOS to process all I/O requests over the NDS-II network instead of local floppies. Due to legal reasons, BDOS and CCP cannot not be changed; therefore, we must modify the BIOS only.

The BIOS can be broken down into three sections:

- A jump table to BIOS functions
- Disk parameter/description blocks
- The code to execute the BIOS functions.

The jump table contains all the entry points into the BIOS functions. The disk parameter section contains blocks that describe how each disk (A:–P:) is formatted, for example, disks can be formatted as single density, double density, or Winchester. The last and largest section of the BIOS is the code that executes the BIOS functions.

In Figure 3, location 0000 hex in the SPA stores the address of the first position (or function) in the BIOS jump table. Therefore, any program running under CP/M may access all the BIOS functions by using the jump vector stored at this address.

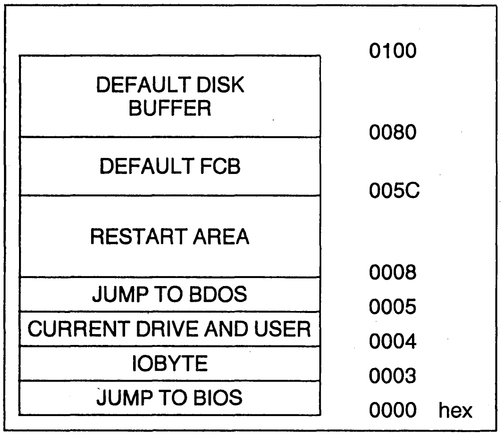


Figure 3. Breakdown of SPA

NOTE:
Any program accessing the BIOS jump vector must access it with a CALL statement, not a JMP statement, since all BIOS functions will use a RET statement to return to the calling program.

The following list of functions are contained in the BIOS and accessed via the BIOS jump table:

- Initial cold start
- Warm boot
- Console status
- Console input
- Console output
- Printer output
- Punch output
- Reader input

Beginning of disk routines

- Home – Restore current disk to A:
- Select disk – select a particular disk to be the current disk
- Select track – select a track in the current disk
- Select sector – select a sector in the current disk/track
- Select disk buffer address – address of where to read/write the sector from/to the disk to/from memory
- Read sector – read the current sector from the current track on the current disk into memory starting at address stored via the BIOS function number 13
- Write sector – write the sector from memory to disk

- List status
- Convert current logical track/sector into real track/ sector

NOTE:
If you are modifying the BIOS, all functions must be supplied.

The Series II standalone version of BIOS currently uses the above functions to access local floppy drives. To modify the BIOS to access NDS-II files, you do not need to change the first eight functions. However, the remaining functions process I/O to the local floppies and must be changed to access I/O over the NDS-II network.

NETWORK CP/M DISK IMAGES

The first problem that must be solved is the accessing of disks. Since Network CP/M does not use the local floppies, a similar storage device(s) must be supplied at the NDS-II file server (NRM).

The NRM Winchester drives are formatted using an extended iRMX™86 structure. This operating system is not similar to CP/M-80. Each NRM drive has a hierarchical file structure and supports multiusers, with individual home directories. Disk storage at the NRM may total four 84 Mb drives. The problem is how to set up a file or disk system on the NRM that will allow CP/M-80 file/disk structures and also support multiuser access to these CP/M-80 disks.

The first design decision in porting CP/M-80 to the NDS-II was to limit Network CP/M-80 to four disks (A:–D:). This decision is two-fold:

- Standalone CP/M-80 only supports A:–D:
- Network CP/M BIOS only has space for four disk parameter blocks.

It was also decided to store these disks at the NRM as data files. These data files will be structured as if they were CP/M floppy disks. Therefore, when CP/M-80 thinks it is accessing a floppy, by track and sector, it will really be accessing a file on the NRM. When a CP/M program wants to read a sector on the disk, our modified BIOS will instead seek into the NRM file to the position where the desired sector is stored and read the sector from the file.

This design creates more problems that must be addressed. The first is the format of the "disk images". The second is the location on these disk images on the NRM to give multiuser support for Network CP/M.

The disk images will be similar to a double density CP/M floppy. Figure 4 shows a layout of a Network CP/M disk image. Network CP/M disks have the following design:

- Each disk has 254 tracks
- Tracks 0 to 3 are reserved for the CP/M operating system
- Track 4 is reserved for directory
- Track size is 2K
- Block size is 2K
- Sector size is 128 bytes
- There are 16 sectors per track
- There is a Maximum of 64 directory entries
- Total disk size is ½ Mbyte

Since the BIOS contains the disk parameters blocks, we must supply four parameter blocks (A:– D:) in the Network CP/M BIOS. These blocks are all identical and describe each disk with the above format.

NOTE:
The Network CP/M utility MAKDSK is used to create disk images. The disk image is created with only the first five tracks (system and directory). As data is added to the disk image, the disk image file will grow by the size of the data. This space-saving feature allows for many CP/M disk images to reside on the NRM file structure, taking only as much disk space as the valid data they contain.

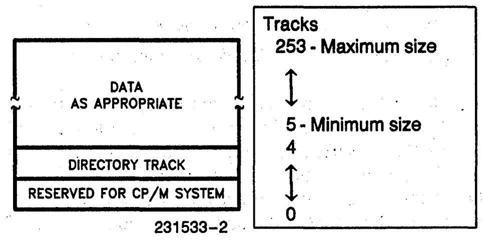


Figure 4. Disk Image Format

Multiuser access to data can be sub-divided into two problems. The first concerns storing all the system files, so that only one copy of the files is on the system, and all users have access to this copy. System files are files/ programs like STAT, PIP, Wordstar, etc. The second problem is that users must have access to private data.

A specific design decision solved these problems. A: disk image will be read only and contain all system files. Therefore, each user on the network will access the same A: disk. B:–D: disks will be private and unique to each user, will store private data, and will have read/write access.

At the NRM all ISIS users access a system directory called ISIS.SYS. This directory is assigned to device :F0: on each workstation and contains all the ISIS system files. Therefore, this is a logical place to store the CP/M disk image A:. The name adopted for the disk image is :F0:ADISK.CPM.

The NRM assigns a directory, called the user's home directory, to each user. This home directory, unique to each user, is assigned to device :F9: on the workstation when the user logs on to the network. Therefore, the home directory is a logical choice to place the disk images B:–D:. The names adopted for B:–D: are:

B:–:F9:BDISK.CPM
C:–:F9:CDISK.CPM
D:–:F9:DDISK.CPM

We have now solved the Network CP/M disk image problem by designing the disk image format and allowing multiusers to access these disk images.

LOADING CP/M OVER ISIS

Normally when a Series II or ISIS cluster is powered on—and boots from—the NRM, the workstation is booted with ISIS. Therefore, our next problem is when running ISIS on the workstation to load CP/M into the workstation memory from disk image A:. We also must be able to restore ISIS when the user is finished with CP/M and wiches to return to ISIS mode.

CP/M-80 is similar to ISIS if both are displayed in their memory map diagrams. Although both operating systems contain the same sections, (see Figure 2 and Figure 5) ISIS and CP/M-80 store the sections in different parts of memory.

In Figure 5, the CLI (command line interpreter) is equivalent to the CP/M CCP region. The ISIS CLI, however, is loaded into the TPA. A command loaded by the CLI starts at address 3680 hex. Therefore, the ISIS CLI is overwritten with the command. When the command or program exits, ISIS will reload and CLI into the TPA region.

Also, a region called the OVERLAY REGION begins at address E800 hex. This area of memory is used by a program that supports overlays. A program that does not require overlays may use memory up to the SPA.

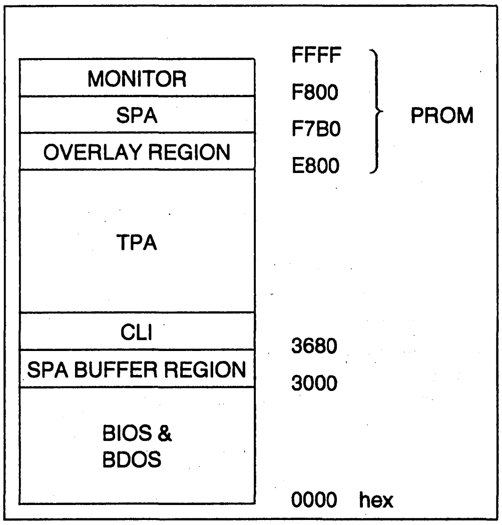


Figure 5. ISIS Memory Map

The problem of how to load CP/M from the disk image A:, saving ISIS, and reloading ISIS when the user is ready to return to ISIS now can be solved. First, create an ISIS program called LOADCPM. This program is a CP/M-80 loader, which:

- Opens file connections to the four CP/M disk images. A: is opened read only; B:–D: are opened read/write access. Note:These disk images are stored at the NRM as normal data files.
- Opens a fifth file called :FO:ISIS.SAV.
- Saves the contents of memory from address 0000 hex to 8FFF hex, and from F000 to FFFF. This is done by writing the contents of the memory into the file ISIS.SAV. This saves the current status of ISIS and the LOADCPM program that is running in the TPA. The ISIS program LOADCPM will not consume the TPA past the address 8FFF; therefore, to save the program the address 8FFF was selected as the upper bound.
- Loads CP/M now that ISIS is saved. The next step is to load the CP/M BIOS, BDOS, and CCP into high memory from the disk image A:.
- Stores the file connections for the disks A:–D: into a BIOS table. This allows the BIOS to read or write to these connections. Therefore, when a CP/M program requests data from disk, the network BIOS will use the file connections to seek into the disk image and read/write the sector.
- Turns control over to CP/M. This is done by calling the first address in the BIOS jump table (cold boot vector). BIOS will first store the return address to the ISIS program called LOADCPM and then turn control over to the CP/M CCP. At this point, the CP/M prompt A will be printed on the console. The user is now running CP/M.

NOTE:
When the last step is finished, the user's workstation (Series II or cluster) is running the CP/M operating system. All CP/M commands/programs are available. Network CP/M is not a CP/M simulator that runs on top of ISIS.

The second part of the solution is how to restore ISIS when the user is ready to return to ISIS. This is done by creating a CP/M program called ISIS.COM. The program operates as follows:

- The program ISIS.COM will reload ISIS and the ISIS LOADCPM program from the ISIS.SAV file.
- At this point, the contents of memory are exactly equal to the point at which the LOADCPM program saved it. ISIS and the program LOADCPM are restored. Now, ISIS.COM will load the PC with the return address to the LOADCPM program, which was saved by the BIOS at the start-up of CP/M. This effectively does a return to the LOADCPM program.
- LOADCPM closes all the disk image files and exits. The exit loads the ISIS CLI, and the user is running ISIS.

Finally, to make the system easier to use, the LOADCPM program is renamed to CP/M. The ISIS user who wishes to run Network CP/M enters the ISIS command: -CPM and when finished, enters the CP/M command: A>ISIS

NETWORK CP/M UTILITIES

Specific ISIS and Network CP/M utilities are supplied with the Network CP/M package. They were developed either as a tool in writing Network CP/M or as a necessary utility needed by the Network CP/M user.

ISIS programs are MAKDSK, ADDSYS, CDIR, CCOPY, CP/M, SUCPM, CPMOMF

Network CP/M programs are ISIS.COM, SPOOL.COM

MAKDSK allows the user to create Network CP/M disk images. The image created will be an empty/non-system CP/M disk.

Command syntax:
```
     MAKDSK <CPM_DISK>
```

Example:
```
     MAKDSK C:
```

ADDSYS allows the user to add the CP/M operating system to the disk image A: created using MAKDSK. This makes the disk image a CP/M system disk. ADDSYS requires that CPM60.COM and BIOS be located in the ISIS.SYS directory on the NDS-II.

Command syntax:
```
     ADDSYS
```

Example:
```
     ADDSYS
```

CDIR allows an ISIS user to list a directory of a Network CP/M disk image or a CP/M-80 diskette in Drive 1 of a Series II with a 720 drive.

Command syntax:
```
     CDIR <CPM_DISK>
```

Example:
```
     CDIR E:
```

CCOPY is a utility that allows an ISIS user to read or write to/from CP/M disk images from/to ISIS files. CCOPY can be used to read data files from the Network CP/M disk images to ISIS files, so that they may be copied to the spooler, etc.

Command syntax:
```
     CCOPY READ <CPM_FILE> TO <ISIS_FILE>
           WRITE <ISIS_FILE> TO <CPM_FILE>
```

Examples:
```
     CCOPY READ   B:MY.DAT     TO :F1:MY.DAT
     CCOPY READ   E:STAT.COM   TO :F1:STAT.COM
     CCOPY READ   A:PIP.COM    TO PIP.COM
     CCOPY WRITE :F1:STAT.COM TO A:
     CCOPY WRITE PIP.COM       TO A:PIP1.COM
     CCOPY WRITE :F1:MY.DAT    TO B:MYSTUFF.DAT
```

CPM is an ISIS program that loads Network CP/M onto the Series II or ISIS cluster.

Command syntax:
```
     CPM [<C_DISK> [ <D_DISKGT ] ]
```

Examples:
```
     CPM
     CPM :F4:BDISK.CPM
     CPM :F4:BDISK.CPM :F5:BDISK.CPM
```

SUCPM is identical to the CP/M utility, except that A: disk is opened for read/write access. This program should have SUPERUSER access rights only. Only one user may run SUCPM. While the user is running CP/M via SUCPM no other users will have access to Network CP/M. The SUPERUSER can use this program to delete files from, or add files to, A: while running Network CP/M.

CPMOMF allows the ISIS user to convert a program developed under ISIS (ie. compiled by PL/M-80 or assembled by ASM-80) to a CP/M executable program. The program compiled/assembled will be written knowing the CP/M environment.

Command syntax:
```
     CPMOMF source_file TO destination_file
```

Example:
```
     CPMOMF SPOOL TO SPOOL.COM
```

SPOOL.COM is a CP/M program that copies a CP/M file to the NDS-II spooler. This utility enables the CP/M user to obtain listings quickly and efficiently. Note: The CPM LST: is not supported in Network CP/M. Listings should be directed to a disk file then spooled to the Network printer.

MODIFYING THE BIOS

At this point, you should have a good idea of how the Network CP/M BIOS accesses NRM disk images instead of local workstation floppy drives. The strategy used to modify the BIOS is the following:

- Use the existing workstation standalone version of the BIOS
- Remove the code used in the BIOS to access the local floppies
- Add the code in the new BIOS to access disk images instead of local floppies.

Due to Intel Corporation Proprietary Information used and contained in the Network CP/M BIOS, the code for the BIOS cannot be released.

intel

ARTICLE
REPRINT

AR-225

August 1982

Debugging Catches up with High-Level Programming

Stuart Vannerson
Electronic Design, June 24, 1982

Reprinted with permission from *Electronic Design* Volume 30, No. 13, copyright Hayden Publishing Co. Inc., 1982.

Order Number: 210671-001

210671-1

Software debugging at the statement and procedure level gives a high-level view of programs from creation to implementation.

Debugging catches up with high-level programming

Although high-level languages for microcomputers have made software design a state-of-the-art procedure, debugging technology has lagged behind. A high-level program debugger brings that technology up to date. By allowing users to monitor and scrutinize PL/M-86, Pascal-86, and Fortran-86 programs at the source level, it addresses some of the key problems faced by high-level language programmers.

The debugger, called Pscope, offers three major improvements over conventional tools:

- High-level debugging at the source statement and procedure level, in addition to the machine level.
- A powerful, reliable code-patching facility, which reduces editing-compiling-linking cycles.
- Symbolic access to all aspects of a user's program, including complex data structures, user-defined data types, dynamic variables, and numerics.

In the past, when microprocessor designs were primarily replacements of simple configurations that used logic gates, the software part of an application was usually written in assembly language. It made perfect sense to debug the application at the machine level, using in-circuit emulators, simulators, logic analyzers, and other discrete tools that worked at the CPU level. However, the increasing size and complexity of microprocessor software has generated a new set of requirements for program debugging.

Although most microprocessor applications today are programmed in high-level language, they employ the debugging tools used for assembly-level programs. In fact, most debuggers reduce high-level language programs to assembly-language equivalents, making debugging more difficult than programming.

An 8086-based software program, Pscope runs on a Series III, Series IV microcomputer development system, XENIX 310, RMX 86, and IBM PC XT, PC AT, along with the user's program being debugged. (It will be used as the software executive for future in-circuit emulators, to combine the benefits of high-level debugging with real time emulation.) Pscope's main advantage is that the user's view of the program during debugging is the same as during its implementation. Stepping, break-pointing, and tracing execution flow are performed on high-level constructs such as statements, procedures, and labels.

Stuart Vannerson, Software Product Manager
Intel Corp.
3065 Bowers Ave., Santa Clara, Calif. 95051

Tracking down bugs

The first thing a designer does once a program has been created, compiled, and linked for execution is to run it. What usually happens is that, due to some logic error, a program takes an incorrect branch and winds up executing in a place it is not supposed to. The designer's first inclination is to find out where that occurred and why.

This is where it is helpful to have some form of trace command. An emulator lets the designer examine the contents of a trace buffer, which gives the past 100 or so CPU instructions executed, plus other information. It even allows disassembly of the contents of the trace buffer. However, if the program went off into some infinite loop, the trace buffer will be filled with just those isolated addresses, and the place where the incorrect branch occurred will have been lost.

The trace facility within Pscope allows setting of trace points at high-level source statements, procedures, and labels. By putting a trace point on each procedure call, for example (as opposed to each CPU instruction), a programmer can look at the trace contents and see exactly the sequence of calls that led to the incorrect branch.

As an example of Pscope's trace capability, consider a program that takes a numeric expression, parses it into tokens, and evaluates it (Fig. 1). By selecting different combinations of its 11 procedure calls to trace, the programmer can change the "granularity" of trace information. While parsing the numeric expression 23 + (19-5*3), and tracing 3, then 7, then all 11 pro-

High-level software debugging

cedures, Pscope generates first 12, then 23, then 74 trace messages, respectively. Tracing CPU instructions, although providing finer granularity, would be much less meaningful in this case.

A programmer debugging at the machine level might try single-stepping to find an incorrect branch. However, just like execution tracing, single stepping is done at the machine level. Working at this level is acceptable when the location of the bug has been determined, but offers little help in finding it. It would take several thousand steps to go through the parsing program in the previous example.

With Pscope, single stepping (like tracing) is done at the source level, using high-level statements and procedures. The LSTEP command executes a program one high-level statement at a time. The PSTEP command does the same, only it treats procedure calls as if the whole procedure were a single statement. In the example (Fig. 2), it takes five PSTEPs to step through the program. In this case, the procedures Sum, Difference, and Maxim were each executed with one PSTEP. If LSTEP were used, the procedures themselves would have been stepped through, and it would have taken 19 LSTEPs. A CPU-level debugger, however, would have stepped through each of the program's 177 machine instructions. It also would have stepped through the run-time routines that were linked in with the program and with the operating system, too. Pscope, in contrast, can differentiate between the user's program module and the run-time routines,

```
SERIES-III Pascal-86, V2.0

  1    1  0  0        program dc (input, output);

                        procedure error(e : error_class); (* print error & restart *)
 40   63  1  1          end (* error *);

                        procedure get_line; (* input line & set c to 1st char of line *)
 60   85  1  1          end (* get_line *);

                        procedure get_token; (* scan line & set t to its value *)

 62   90  1  0            function digit(c: char): boolean; (* true if c is a digit *)
                          end;

 64   95  1  0            function upper_case(c: char): boolean; (* true if c is upper case *)
                          end;

 66  100  1  0            function lower_case(c: char): boolean; (* true if c is lower case *)
                          end;

                          procedure get_char; (* set c to next char in line *)
 79  119  2  1            end (* get_char *);

                        begin (* get_token: scan line & set t to its value *)
135  170  1  1          end (* get_token *);

                        procedure factor(var factor_value : integer);
139  177  1  0          begin (* factor *)
168  202  1  1          end (* factor *);

                        procedure term(var term_value : integer);
173  210  1  0          begin (* term *)
188  223  1  1          end (* term *);

                        procedure expression (var expression value : integer);
193  231  1  0          begin (* expression *)
221  254  1  1          end (* expression *);

                        procedure statement;
224  260  1  0          begin (* statement *)
228  264  1  1          end (* statement *);

229  257  0  0        begin (* main program *)
                        repeat (* forever *)
232  278  0  2            get_line;
233  279  0  2            get_token;
234  280  0  2            statement;
235  281  0  2          until false;
238  237  0  1        end.
```

(a)

210671-3

High-level software debugging

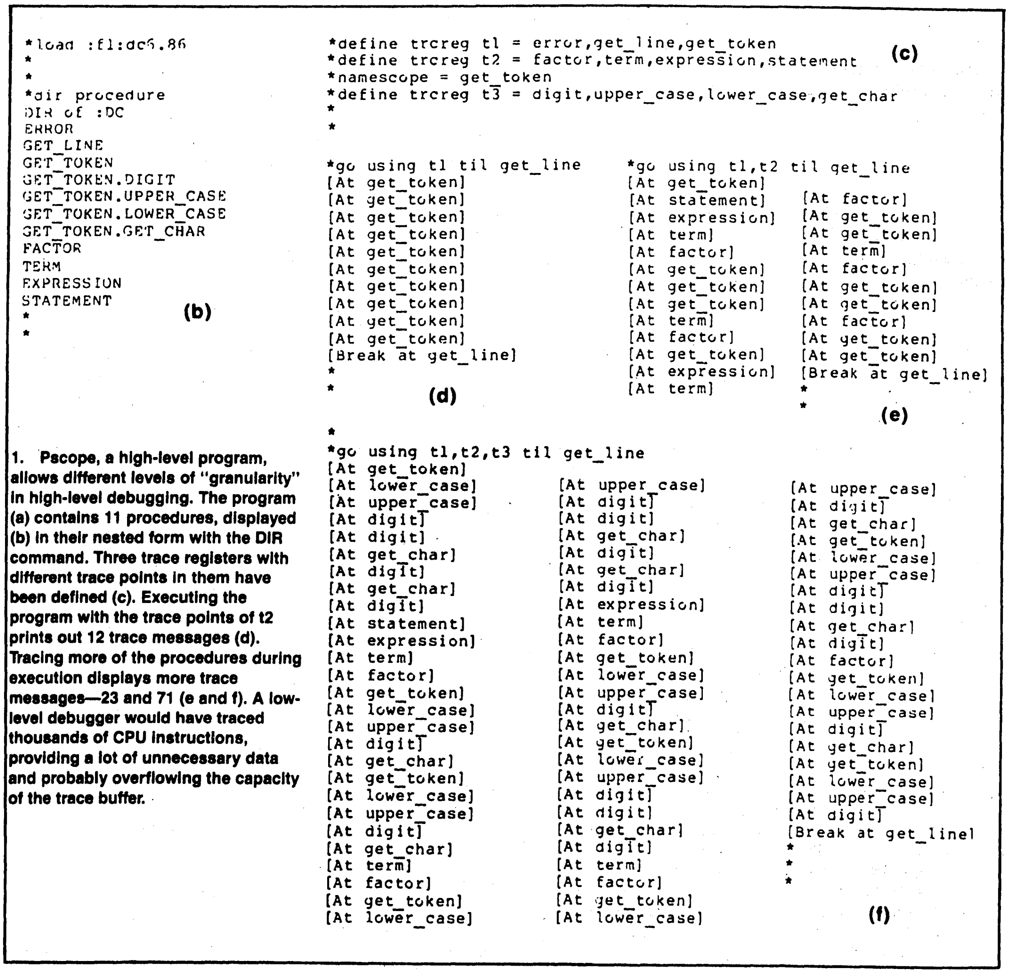
eliminating a lot of tedious or wasted effort.

All of Pscope's commands allow users to examine and manipulate a program (both code and data) using the same symbolic constructs in which the program was developed. Breakpoints, like trace points, may be placed at procedure calls, procedure returns, statements, and labels. Thus, to debug a program with Pscope, all a programmer needs is a listing. Linkage maps, memory dumps, locate maps, and the like are unnecessary. In addition, the number of high-level breakpoints (and trace points) is unlimited with Pscope. On the other hand, since emulators use hardware breakpoints, they are limited to just a few.

Now suppose that the programmer has determined the specific procedure where the program took an incorrect branch. The next job is to find out why. Jumps and calls are usually based on the outcome of a particular condition—if the condition evaluates true, the program goes one way; if false, another. The conditions are often complex, however; they may involve several Boolean expressions and elements of some obscure data structures.

To track down the cause of the bug, a programmer begins examining the contents of these data elements, compares them with what he or she expected them to be, and moves a step further. This procedure usually involves stepping a statement at a time and looking at data values.

Unfortunately, the traditional low-level debuggers provide only symbolic access to variables of some basic



```
*load :f1:dc6.86
*
*
*dir procedure
DIR of :DC
ERROR
GET_LINE
GET_TOKEN
GET_TOKEN.DIGIT
GET_TOKEN.UPPER_CASE
GET_TOKEN.LOWER_CASE
GET_TOKEN.GET_CHAR
FACTOR
TERM
EXPRESSION
STATEMENT
*
*
```
(b)

```
*define trcreg t1 = error,get_line,get_token
*define trcreg t2 = factor,term,expression,statement
*namescope = get_token
*define trcreg t3 = digit,upper_case,lower_case,get_char
*
*
```
(c)

```
*go using t1 til get_line
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[At get_token]
[Break at get_line]
*
*
```
(d)

```
*go using t1,t2 til get_line
[At get_token]
[At statement]       [At factor]
[At expression]      [At get_token]
[At term]            [At get_token]
[At factor]          [At term]
[At get_token]       [At factor]
[At get_token]       [At get_token]
[At get_token]       [At get_token]
[At term]            [At factor]
[At factor]          [At get_token]
[At get_token]       [At get_token]
[At expression]      [Break at get_line]
[At term]            *
                     *
```
(e)

```
*
*go using t1,t2,t3 til get_line
[At get_token]
[At lower_case]      [At upper_case]      [At upper_case]
[At upper_case]      [At digit]           [At digit]
[At digit]           [At digit]           [At get_char]
[At digit]           [At get_char]        [At get_token]
[At get_char]        [At digit]           [At lower_case]
[At digit]           [At get_char]        [At upper_case]
[At get_char]        [At digit]           [At digit]
[At digit]           [At expression]      [At digit]
[At statement]       [At term]            [At get_char]
[At expression]      [At factor]          [At digit]
[At term]            [At get_token]       [At factor]
[At factor]          [At lower_case]      [At get_token]
[At get_token]       [At upper_case]      [At lower_case]
[At lower_case]      [At digit]           [At upper_case]
[At upper_case]      [At get_char]        [At digit]
[At digit]           [At get_token]       [At get_char]
[At get_char]        [At lower_case]      [At get_token]
[At get_token]       [At upper_case]      [At lower_case]
[At lower_case]      [At digit]           [At upper_case]
[At upper_case]      [At digit]           [At digit]
[At digit]           [At get_char]        [Break at get_line]
[At get_char]        [At digit]           *
[At term]            [At term]            *
[At factor]          [At factor]          *
[At get_token]       [At get_token]
[At lower_case]      [At lower_case]
```
(f)

1. Pscope, a high-level program, allows different levels of "granularity" in high-level debugging. The program (a) contains 11 procedures, displayed (b) in their nested form with the DIR command. Three trace registers with different trace points in them have been defined (c). Executing the program with the trace points of t2 prints out 12 trace messages (d). Tracing more of the procedures during execution displays more trace messages—23 and 71 (e and f). A low-level debugger would have traced thousands of CPU instructions, providing a lot of unnecessary data and probably overflowing the capacity of the trace buffer.

210671-4

High-level software debugging

program types. Complex structures, user-defined data types, stack-based dynamic variables, and numerics all are examples of data that requires more complex handling. For example, to access a field within a record using the ICE-86A emulator, users must give the byte offset from the beginning of the record (e.g., user_rec + 14). On the other hand, Pscope allows the designer to access fields by name, for example, user_rec. age. Representation of floating-point numbers requires binary-to-decimal conversion, a luxury many debuggers leave off. Pscope lets a designer examine and modify real numbers at three precision levels, providing conversion from binary into decimal back into binary. Examination and modification of all data is done symbolically in Pscope.

The advantage of all this is that data references are easier, and fewer mistakes are made (the designer does not have to calculate offsets). Thus the process of stepping, looking at data, evaluating expressions, and continuing are speeded up. In other words, bugs are tracked down faster.

Fixing bugs

Tracking down the location of bugs quickly is only half the battle. Correcting the problem is the other, time-consuming half.

For large applications the program-change cycle can be lengthy. Program changes are made with an editor; then the source is recompiled and the module linked with the run-time routines and other modules. Since programs can initially contain a lot of bugs, going through an editing-compiling-linking cycle for each bug can become extremely wearing after a while.

```
*pstep
[Step at :EXAMP#21]
*pstep
                INPUT TWO INTEGERS:
[Step at :EXAMP#22]
*pstep
                    (input)  319 46
[Step at :EXAMP#23]
*pstep
                    THE SUM IS  365
[Step at :EXAMP#24]
*pstep
             THE DIFFERENCE IS  273
[Step at :EXAMP#25]
*pstep
                THE MAXIMUN IS  319
                THE MINIMUM IS  46
[Step at :EXAMP#21]
*
```

2. This program illustrating the stepping features of Pscope, contains five statements in the main body, three of which are procedure calls. Five procedure steps are required to go through the program. It would have taken 19 statement-level steps, as each procedure would have been stepped though. In contrast, a CPU-level debugger would have stepped through all 177 instructions, as well as through the run-time system.

Programmers typically go to great measures to avoid such a necessity. Instead, they often try to patch the object module, in order to continue debugging.

Patching object code, however, may be difficult for a variety of reasons. First of all, the desired patch must be written in hex code or assembler mnemonics. Those debuggers that disassemble object code usually offer a line-by-line assembler as well. Patching a high-level program at the machine level can be a horrendous mess, though. The high-level language compiler may have adopted certain stack conventions, code optimizations, and register usage that make it difficult to understand what to patch, let alone how to patch it.

The patch is frequently a different size than the code to be patched, and that introduces more complications. An unfortunately common solution is to jump to an unused location, perform the patch, and jump back to the return address. Another problem is that even though the machine-level patch may work, incorporating the change into the source file later may generate entirely different code from that of the patch. Because of all these complications, patches are used only in simple cases, where programmers can easily determine the results. It is unfortunate, too, because a good patching mechanism could eliminate a lot of programmers' headaches.

In lieu of machine-level patching, the common methodology is to set a breakpoint at the location of the bug and correct the problem by hand. Correcting the problem usually means reassigning variables or reversing the outcome of some IF . . . THEN conditions. These methods are simpler than patching but introduce problems of their own: If the bug is located inside a loop, the "breakpoint and change by hand" approach must be done too frequently. Also, if the manual changes are more than a few assignments, the process becomes tedious. Lastly, only a few bugs can be changed in this fashion before it becomes difficult to keep track of them. As a result, programmers quickly resort back to extensive editing-compiling-linking cycles.

Pscope's approach to the problem is to provide an automated way of writing and managing high-level code patches. Rather than define changes to the program at the machine level, users may define code patches at statement numbers. With Pscope, the actual contents of the patch may be complex as well—DO . . . END blocks, IF . . . THEN . . . ELSE conditions, and REPEAT . . . WHILE . . . UNTIL loops make the command language as powerful as Pascal or PL/M.

Using high-level code patches is simple (Fig.3). After determining the location and cause of a particular bug, the programmer defines a patch. In this example, a multiplication took place at line 5, rather than an addition. The command define patch #5 til #6 = z=x+y causes the contents of the patch to be executed in place of the statement at line 5. The designer can

High-level software debugging

specify any starting point and any point to continue execution. Furthermore, patches are executed in all GO, LSTEP, and PSTEP commands without having to specify them. Perhaps the biggest help in managing patches is that it is easy to see where they are (DIR PATCH); in addition, they may be written out to disk (PUT filename PATCH). Thus, it becomes very easy to incorporate them into the source file later. Because the patch language is so similar to the source language, a patch that worked in Pscope is most likely to work in the modified program as well.

Many utility commands will be part of most simple debugging sessions. Pscope's "literally" feature allows users to abbreviate, redefine, and tailor the command language to suit their needs. For example, define literally d = 'define' lets the programmer use d for the define command. The HELP command displays (on the console) the usage and syntax of most commands and facilities in Pscope. The PUT and INCLUDE commands let users write and retrieve commands (usually definitions of break and trace registers, program patches, and "literally's") on disk, to use in later Pscope debugging sessions.

Pscope's command language is a powerful progamming language that is used for generating new commands (debugging procedures), the same way high-level code patches are defined. Debugging procedures allow you to define compound and conditional commands. Like procedures in high-level language, these procedures may have parameters, may supply return values, and may have their own local variables. Thus Pscope is in fact a programming language in its own right.

Debugging procedures may be called automatically upon reaching a breakpoint or a trace point. When a breakpoint is reached, Pscope can call a procedure that contains any sequence of Pscope commands. Conditional break and trace points may be set up this way. By evaluation on condition in the procedure, a return value of "true" or "false" determines whether the break (or trace message) should take place or not.

Debugging procedures (and code patches) are created with Pscope's built-in editor and may be stored on disk. The editor is a menu-driven, CRT-oriented program that is used to edit not only debugging procedures, but command lines as well. For example, when a syntax error occurs on a long command line, the user just hits < esc > on the keyboard and the editor comes up. The command will then be reexecuted when it is corrected.

The command language has conditional constructs (IF... THEN... ELSE), looping control (REPEAT... WHILE ... UNTIL ... COUNT), calls and returns for procedure nesting, and a full set of program data types. The data types correspond to the recognized types of the user's program (PL/M and Pascal). Debugging procedures can also access user-program variables (for example, debug_variable = prog_count + t).

These procedures also allow program stubs to be expanded. Rather than resolve external program references with fully coded modules, subsystems can use empty stubs for resolving externals. During debugging, then, a procedure can be used to supply input values, take outputs and process them, supply return values and conditions, and so on. In essence, procedures can be set up so that all or part of a software system is modeled. Such flexibilty affords much greater independence in software implementation, as separate software modules can be developed and then debugged independently.

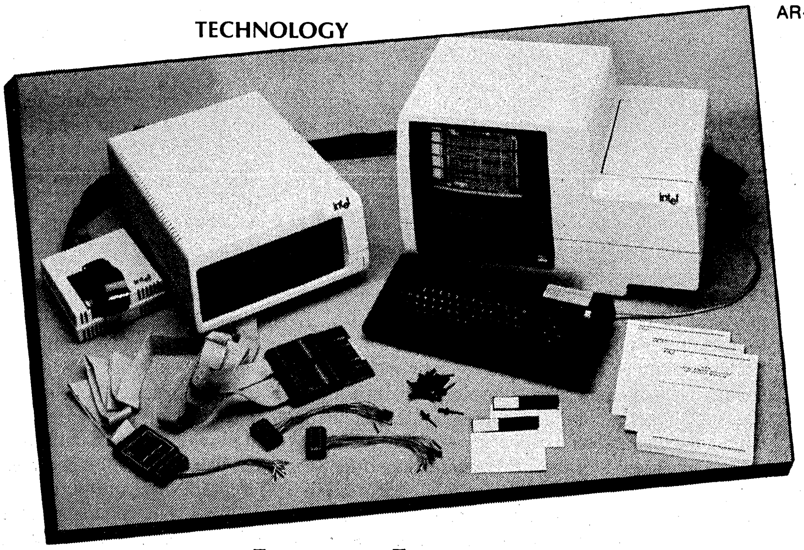
Lastly, debugging procedures can be used to automate the software testing process. Complete (or incomplete) systems may be executed over and over again, each time with new parameters and each time recording the results. The parameters can be selected by the designer or derived algorithmically using debugging procedures.□

```
*load :fl:maxmin.86
*
*
*go til #21
[Break at :EXAMP#21]
*go
               INPUT TWO INTEGERS: 19 4
                   THE SUM IS  76
           THE DIFFERENCE IS  15
               THE MAXIMUN IS  19
               THE MINIMUM IS   4
[Break at :EXAMP#21]
*
*
*  /*  looking at statement #5 in
**     the program, notice we multiplied
**     instead of added.  Let's patch it   */
*
*
*define patch #5 til #6 = z=x+y
*
*go
               INPUT TWO INTEGERS:  19 4
                   THE SUM IS  23
           THE DIFFERENCE IS  15
               THE MAXIMUN IS  19
               THE MINIMUM IS   4
[Break at :EXAMP#21]
*
```

3. High-level code patching can fix the bug in statement 5, which calls for multiplying, instead of adding, two integers. A patch is defined to replace this statement, and the program now executes correctly. Patches remain active during all LSTEP, and PSTEP, and GO commands until the patch is removed.

210671-6

TECHNOLOGY



Integrated software-development instrumentation *can significantly reduce the development costs involved in bringing a product to market. The Intel I²ICE system comprises an in-circuit emulator for 16-bit microprocessors, a logic-timing and state analyzer and a high-level language debugger connected to a host development computer.*

Software development

PAUL MARITZ, Intel Corp.

New tools and approaches are boosting software-development productivity

The past year has seen a transformation in software development for microprocessor systems, involving more sophisticated processors, increased software content in the end product and a growing shortage of software talent. The integration of common human interfaces across heterogeneous systems, coupled with a tremendous focus on "friendly" and "productivity-based" features and the incorporation of classic hardware tools, such as in-circuit emulators, into the software-development environment have changed the very structure of the software lab. While in 1982 the concept of an integrated workstation combined an emulator with a logic analyzer, in 1984 an integrated workstation will combine software tools with hardware assistance to boost software productivity.

The cost to a company of a malfunctioning or poorly designed product can prove far more expensive than doubling its R&D expenditures or absorbing a significant increase in the product's cost. This is equally true for the software-development process. "Time to market is everything," and this consideration will become significantly more important over the next few years.

Increasing software productivity

During 1984, changes in computer systems will continue the evolution outlined above. Software tools will become available to guide the documentation and building of software systems, hardware will help software engineers evaluate software "completeness," and performance analysis and high-quality local-area networks (LANs) will be pervasive in every medium and large software environment. Just as logic analyzers, oscilloscopes and emulators have assisted hardware engineers, documentation aids, very high-performance distributed processing and the adaptation of emulators to software-intensive environments will lead the way to a greater measure of software productivity in the mid-1980s.

The key to software productivity lies in minimizing—or eliminating—a focus on learning to use the tools and maximizing the development and convenience of common human interfaces, high-level software tools and automated documentation and software-version control. No matter how well each individual tool works in and of itself, the effectiveness of the design aids available to the software engineer depends more on the interaction and interdependence of each tool than on any one tool's features.

The single most time-consuming task in the software-development cycle (Fig. 1) is verifying that the software works—that is, detecting and correcting bugs. One reason this process is so inefficient is that debugging is done at a low level. Most programs today are written in a high-level language. A software into low-level, processor-specific (object) code. The compiler could pass information about the program to the debugger, so that it could present the software engineer with information about the program in high-level terms (Fig. 2). This is an example of a human interface that is efficient, not just friendly.

The cost to a company of a malfunctioning or poorly designed product can prove far more expensive than doubling its R&D expenditures or absorbing a significant increase in the product's cost.

In microprocessor development, it is often necessary to complete the verification of the software by running the program in the target environment—the real-world environment of the application to be controlled. This is usually a necessary step because the interaction of the microprocessor and its external environment might be very complex and exceedingly difficult to simulate. For example, consider a microprocessor controlling a robot arm. The microprocessor must receive instructions on where to move the arm and, at the same time, monitor sensors reporting the state of the motors driving the arm. These inputs arrive in an unpredictable sequence and must be serviced within certain time limits if the robot arm is to perform as expected.

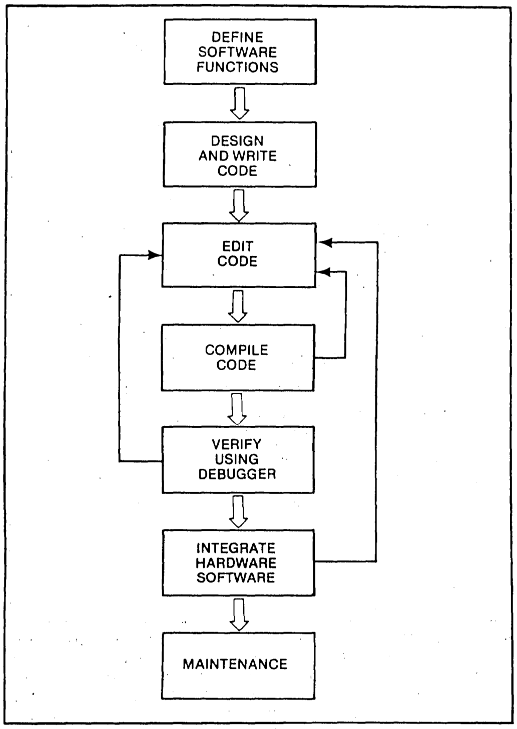


Fig. 1. In a typical software-development cycle, *problems in compiling the code, verifying it with a debugger and integrating hardware with software send a project back to the editing stage. Problems become more difficult to correct as development proceeds and particularly difficult to rectify after hardware is involved. A project that fails to use the appropriate tools throughout its life cycle risks slipping all the way back to the definition and design/writing phase.*

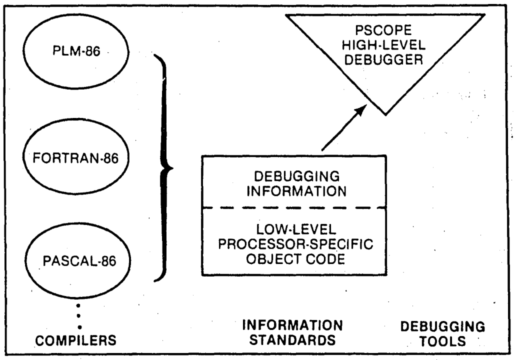


Fig. 2. Using a high-level debugger, *such as Intel Corp.'s PSCOPE, in the software-development process allows a developer to correct problems with code in a high-level language instead of in low-level, processor-specific terms. Such tools can greatly increase the efficiency of the debugging process.*

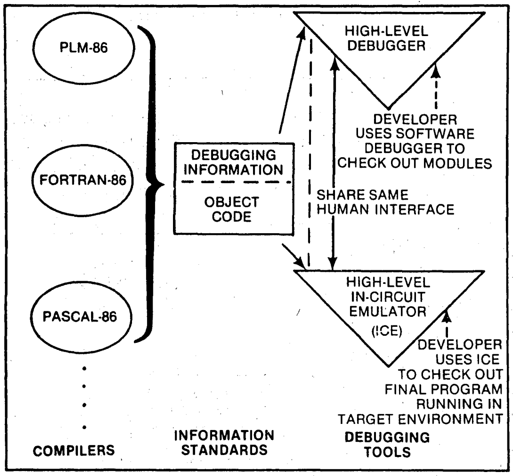
engineer uses a language translator that translates high-level terms and constructs that are closer to an application into low-level or processor-specific terms. For example, a programmer might write his program in Pascal, considering such entities as procedures, records and expressions. However, when he detects a bug in his program, he is forced by the available tools to operate at the processor level and to deal with such entities as hex numbers, registers and flags. Because the programmer has to translate manually back from a low level to a high level, productivity is lost.

Implementing high-level debugging

Why not, instead, have the debugging tool do a reverse translation? After all, the programmer submitted the high-level description (source code) of his program to a translator (compiler) to have it converted

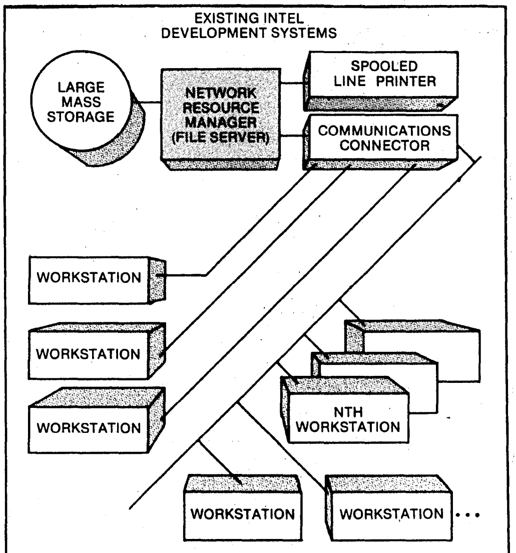
Simulating such an environment would be as much trouble as writing the target program. The ideal approach therefore involves simulating only those program modules that have a well-defined and simple input and output sequence and hence can be debugged easily in a simulated environment. The complete program, with its complex, time-dependent inter-module relationships, can then be debugged in the actual target environment.

This two-stage approach requires two types of related debuggers: a software debugger that allows the software developer to simulate modules on his workstation and an in-circuit emulator that allows him to debug the software running in the target environment. To be most effective, these two types of debuggers should share the same human interface (Fig. 3), permitting an engineer to move easily from module-level simulation to in-target debugging without mentally shifting gears.



PLM-86
FORTRAN-86
PASCAL-86
COMPILERS
DEBUGGING INFORMATION
OBJECT CODE
INFORMATION STANDARDS
HIGH-LEVEL DEBUGGER
DEVELOPER USES SOFTWARE DEBUGGER TO CHECK OUT MODULES
SHARE SAME HUMAN INTERFACE
HIGH-LEVEL IN-CIRCUIT EMULATOR (ICE)
DEVELOPER USES ICE TO CHECK OUT FINAL PROGRAM RUNNING IN TARGET ENVIRONMENT
DEBUGGING TOOLS

Fig. 3. High-level software and hardware debuggers *(yellow) sharing the same human interface speed software developmeht. A developer can use a high-level debugger to exercise a software module on a workstation before all the modules of a program are available or before a prototype target system is constructed. When all modules and the prototype are ready, an in-circuit emulator, such as Intel's 12ICE, can be employed to debug the code running in real time in its real-word environment. Using a variety of compilers allows the developer to choose the optimum language for each sub-task and, in many cases, to use off-the-shelf software components written in a standard language.*



COMMUNICATIONS PROTOCOLS USED BY NDS II

Layer in ISO model	Protocol used today	Future evolution
upper level protocols	Intel network architecture	public standard
transport protocol	Intel network architecture	ISO standard
physical and data link protocol	Ethernet	Ethernet/IEE-802

Fig. 4. An LAN can integrate shared and dedicated software-development resources. *A shared, central database allows the storage and management of project information. Dedicated, single-user workstations provide team members with processing power, large memory space and quick response. An LAN linking individual workstations furnishes communications between software developers and common access to database information through a network resource manager. An efficient LAN, such as Intel's NDS-II, will eventually be able to connect workstations from different manufacturers to serve changing software-development needs. This goal, however, requires further standardization of communications protocols, the aim of the International Standards Organization (ISO) and other groups.*

Managing software development

Typically, programmers generate at least three classes of information: documentation, source and object (processor-specific) code. More than one individual usually generates the information produced by a development project. In addition, the information usually undergoes changes over time, resulting in several different versions of the software. Furthermore, a typical project involves many variants of the information produced, such as one for floppy disks or one for Winchester disks.

On a multiengineer software-development project, the management of these different levels of information can become problematic. And, although the cause of the problems is usually simple, their effect is very costly. An engineer might waste days building a test system with an out-of-date document. Another problem that frequently arises is that of a "mysterious" change—an engineer changes a module and then fails to notify others of the modification.

Although these are simple management problems, a week lost on a 10-man engineering project because of an incorrect source file can mean $20,000 to $30,000 in direct staff costs and a serious slip in the product-development schedule.

Solving development problems

Automating software-management procedures can solve these types of problems by providing project members with a database in which to hold and control project information. It can also furnish the software-generation tools needed to build the correct software systems from information held in the database (see "Automating software management," below).

A project database should be able to provide:

- automatic separation of information according to type, version and variant. For example, a user must be able to extract from the database "the source associated with module X, version 2—the floppy disk variant" or "the test data for module Y, version 3."
- control of access to information. A software developer must be able to lock, or "freeze," certain versions to prevent problems arising from mysterious changes to the base information.
- a guaranteed audit trail for all information—what changes were made, by whom, why and when—making it easier to track the changes made in going from "version 2.0 to version 3.0."

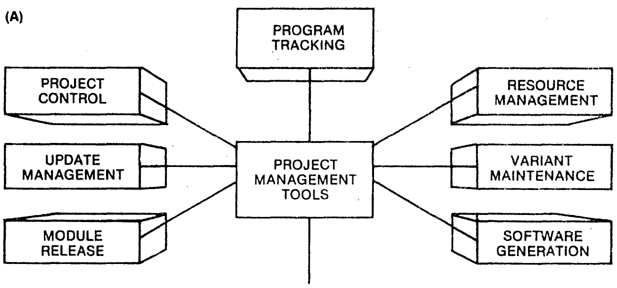
A software engineer should also be able to specify the desired software mix for the end product: the modules to be compiled and linked, the versions and variants to be used and the modules that rely on each other. From this description, a utility can extract the correct information modules out of the database and compile, assemble and link the source and object files to create up-to-date, consistent software. Ideally, the utility should be able to avoid redundant steps if the required information already exists. For example, there should be no need to recompile the source of module X as long as a good copy of the object for module X exists in the database. A single change in one module should not require the recompilation of all 60 modules in a project.

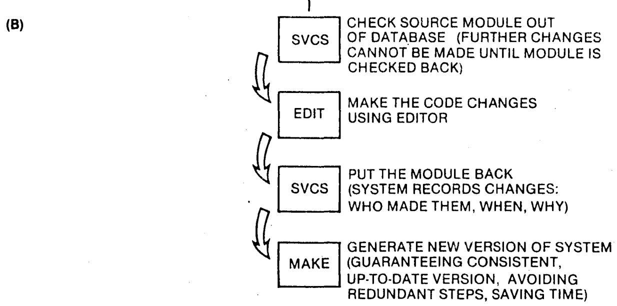
Shared resources + dedicated resources

In today's software-development environment, two conflicting requirements are placed on host systems. First, software developers must be able to communicate and *share* resources. Information-management tools typically require that members of a project share a central database in which project information is stored and managed. Software engineers must be able to communicate information, performing such functions

AUTOMATING SOFTWARE MANAGEMENT

Creating a new software product is a complex, multistage process usually involving many software-development team members, three kinds of information and code (documentation, source and object code) and several versions and variants of a package. For example, developing an Intel Corp. compiler for the 8086 microprocessor involved 175 modules totaling 200K bytes of code, four engineers and 10 hours of program-generation time. Correctly managing such a project and avoiding costly mistakes increasingly requires automated project-management tools (A), such as Intel's Software Version Control System (SVCS) and MAKE. The SVCS system furnishes a database that permits project members to track and control versions and variants of the software. Database information can be protected against accidental or simultaneous changes by two or more developers. The system can also record when a change in a software module was made and the reason for and initiator of the change. The MAKE facility can determine what compiles, assembles and links must be performed on various modules to construct a software product from its constituents. The utility uses module-dependency information (what modules affect certain other modules) to ensure consistent, updated software and eliminate redundant steps. A programmer, for example, can use these project-management tools to alter a program module, put the module back in the system and generate a new, consistent version of the system (B).




(B)
SVCS
CHECK SOURCE MODULE OUT OF DATABASE (FURTHER CHANGES CANNOT BE MADE UNTIL MODULE IS CHECKED BACK)
EDIT
MAKE THE CODE CHANGES USING EDITOR
SVCS
PUT THE MODULE BACK (SYSTEM RECORDS CHANGES: WHO MADE THEM, WHEN, WHY)
MAKE
GENERATE NEW VERSION OF SYSTEM (GUARANTEEING CONSISTENT, UP-TO-DATE VERSION, AVOIDING REDUNDANT STEPS, SAVING TIME)

as sending and receiving electronic mail. This trend favors the use of minicomputers that let users share a database, communicate and cooperate.

Second, software developers' tools are becoming increasingly sophisticated. The price of this sophistication, however, is more powerful computing resources. These tools require *dedicated* processing power, large memory space and quick response to perform efficiently. This means newer tools will have to be hosted single-user workstations, in which software developers can be guaranteed a certain level of computing resources.

Single-user workstations connected to a local-area network (LAN) can resolve these conflicting trends (Fig. 4). Such a distributed-processing approach offers the best of both worlds. Each user has a dedicated set of computing resources in his workstation, uses a central, shared database located at the file server and can easily communicate with other users.

If the LAN architecture is correctly designed, distributed processing can offer other benefits as well. Different types of workstations can be attached to the network, according to user needs. Thus, in a software-engineering environment, most of the workstations could be optimized for software developers, with only a few reserved for hardware debugging. To meet these needs, the LAN should become the standard information bus of the software-development team.

The distributed processing afforded by an LAN will also provide a measure of protection against obsolescence. Newer stations can be added to replace older ones, as required. Now, the unit of growth for software-development systems is the workstation, not the mainframe.

The trend toward workstations connected by an LAN weighs heavily against the cost advantage of timesharing over distributed processing. The push for software productivity will therefore be the most pressing reason for the adoption of distributed processing in modern software-development environments. □

Paul Maritz is software tools planning chairman at Intel Corp.'s Development Systems Operations, Santa Clara, Calif.

Reprinted from **MINI-MICRO SYSTEMS** December 1983

intel®

ARTICLE
REPRINT

AR-352

Integrated Environment Speeds System Development

Kenneth Pomper
Dennis Carter
Intel Corporation
Santa Clara, California

Reprinted from *Computer Design*, November 1984

ORDER NUMBER:231257-001

231257-1

INTEGRATED ENVIRONMENT SPEEDS SYSTEM DEVELOPMENT

By integrating source and version control, electronic mail, standard interfaces for programming languages, and common interfaces to operating systems, a total development environment can accelerate the software task faster than adding staff.

by Kenneth Pomper and Dennis Carter

If a project is running behind schedule, adding staff members is not always the best tactic for getting it back on schedule: as the saying goes, adding manpower to a late software project makes it later. Often the best solution is to coordinate programming efforts and project management through an integrated development environment. This type of system stimulates greater efficiency by combining management, programming, and debugging tools in one environment. Productivity increases especially with microprocessor systems with separate target and host development systems. As a result, industries can meet critical delivery schedules without needing additional programmers.

System development is a complex process involving several different stages that continually pass information between each other. The development environment should be more than a collection of assorted tools that are poorly linked. It must efficiently coordinate the diverse stages of development in a single coherent environment, allowing information to flow easily between different tiers of the project (Fig 1).

An efficient development cycle has two parts. Managers must have a clear view of the project from inception through test and implementation. Thus, planning

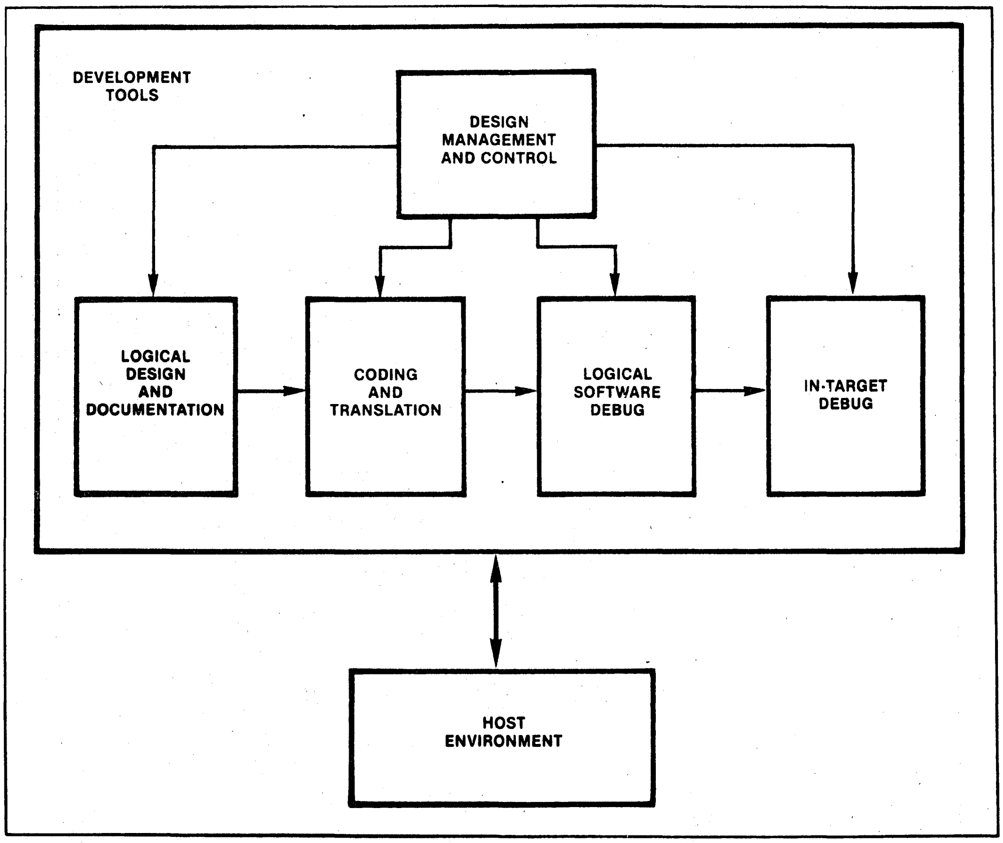


Fig 1. An integrated development environment must do more than act as a library for development tools. It must ensure that information flows smoothly between components. As organizations shift to new development policies and expand development hardware, the system must be able to migrate smoothly to the new host environment.

231257–2

work schedules and anticipating design bottlenecks is easier. Software engineers must share their ideas, designs, and programs, passing information throughout the different development stages.

Yet, in developing products for other target machines, an integrated environment for the host development system alone is not enough. Unless a smooth transition to the final target environment is provided, the project will bog down during the critical target system integration and test. The transition from host to target development environments is one of the two major factors affecting the project cost. According to R.W. Jensen, changing environments can increase costs as much as 122 percent.

Not only must engineers deal with different target hardware in different projects, but also they must work on a shifting host hardware base as companies expand their development resources. Rather than losing previous investments in tools or training, the company must be able to shift the entire environment smoothly as the company shifts to different development strategies. For example, engineers using Intel's Intellec® Series IV workstation maintain the same fundamental development environment when they move to the NDS-II distributed development environment.

With its multiple stages, development can turn into a logistical headache for managers and engineers alike. Managers supervising several programming teams, each developing different versions of programs, can easily lose the thread of revisions to source code. Similarly, programmers can find themselves working at cross-purposes in their attempts to generate and test the most recent versions of code, rather than a hybrid of current and obsolete code versions.

An integrated system can help prevent these problems by combining different tools and making them work well together. For example Intel's configuration management tools, Source Version Control System (SVCS) and MAKE, manage multiple versions of a program. The tools can automatically combine the most current versions of several modules in larger programs. Similarly, Intel's debugging aids, PSCOPE and Integrated Instrumentation and In-Circuit Emulation (I^2ICE^{TM}) package, use information implanted by compilers to permit programmers to debug during the integration process at the source-level. Such an integrated environment increases efficiency through good allocation of available resources.

MANAGEMENT AND CONTROL

Modular design helps software engineers break a large complex problem into a set of small simple programs. Unfortunately, a modular design system requires more overhead for managing a large number of modules and different versions of the same module. If the logistics become too troublesome, programmers might even collapse several modules into a single file to save themselves the trouble of manipulating the separate modules. Project management tools can free engineers from the housekeeping chores associated with program development (Fig 2).

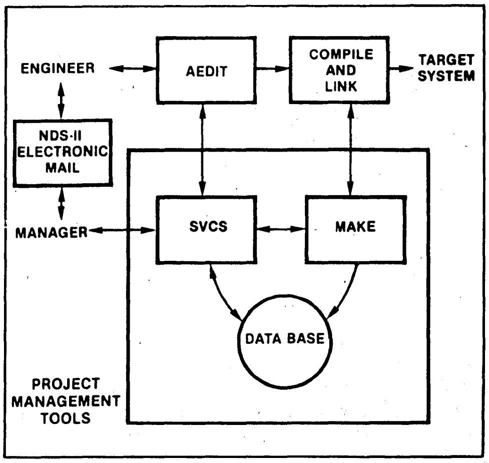


Fig 2. Besides controlling changes to the source files in its data base, SVCS helps managers audit source updates. Automatically generating the software for the target system, MAKE reduces generation time by about 50 percent, leaving engineers more time to concentrate on development.

Programmers keep track of major changes in their programs by either creating copies of the new version or changing an older version. The result is a series of similar programs that lack proper documentation to indicate the change and reason for the change. SVCS provides an automated approach to this record keeping. It tracks changes to the baseline version of a program, and demands that programmers record their reasons.

When software engineers need a particular version of a file, whether the current or some older copy, SVCS automatically retrieves the correct version from its data base of updates and baseline versions. Similarly, after the programmers have added changes, SVCS records the updates and the reasons for the changes, adding as little as a 3 percent overhead. In addition, SVCS helps project managers exercise precise control in large team projects by preventing certain engineers from making changes independently.

231257-3

While programmers work directly with SVCS to manage different versions of programs, MAKE works closely with SVCS facilities to generate current versions of systems. While generating large systems from several different modules, programmers often find that one or two modules have been updated since the last compilation. This problem is compounded when modules depend on a series of other submodules. MAKE automates the manual procedures often resorted to by software engineers to track current object modules.

Using templates that detail the modules' interdependence, MAKE ensures that only current versions of modules are included in the system generation. If it finds that a required object module is obsolete, MAKE automatically compiles the appropriate source module to produce the current version of the object module. Furthermore, if source modules depend on submodules, MAKE continues searching through its templates to ensure it recompiles modules using the current submodules for these source modules.

MAKE selectively compiles the needed modules. Only if a module or one of its submodules is obsolete does MAKE execute a recompilation. This cuts the inefficient massive compilation procedures commonly used to ensure that object modules are current.

In addition to the project management tools handling version control and system generation, a complete integrated development environment should also facilitate communication among users. Acting as an electronic central distribution center, the NDS-II electronic mail facility maintains mailboxes for individual users and groups of users on the network, and an electronic bulletin board for all users. In addition to supporting document distribution, electronic mail manages a file transfer facility. Team members can transmit both source and object modules to any other user on the network.

Another feature, NDS-II's network resources manager (NRM), provides extensive support for file management and resource sharing. The NRM manages files with a hierarchical structure that arranges files into volumes and multiple subdirectories. The NRM also improves allocation of resources through its distributed job control (DJC) facility. DJC permits users on private workstations to export a batch job to the NRM for remote execution. The NRM then moves the job to a free workstation for execution, returning the completed job status to the user's directory.

LOGICAL DESIGN

An integral part of the software development environment and its primary interface with the user is the text editor. Because software engineers typically spend 40-50 percent of their time using a system editor, it is a critical element in software development and can greatly enhance productivity if used well. For example, programmers often need to work simultaneously on two separate files, such as two different source programs or a program and a specification document. Editors such as Intel's AEDIT permit them to edit two files of any size simultaneously and transfer text between them.

AEDIT's ability to store a sequence of edit commands also simplifies the use of edit macros. With AEDIT, programmers build macros simply by typing in their commands. They can re-execute the command series and even save it on disk for later use. AEDIT also helps software engineers with structured programming techniques through its automatic text indentation. Furthermore, AEDIT protects programmers' efforts by optionally creating back-up copies of files being edited.

Although a text editor serves as the primary interface between the development system and programmer, programming languages serve as the principal interface between design concepts and the target hardware. With the right set of programming languages and support tools, software professionals can develop the optimal solution for a particular situation, without the design bias often seen when designers plan projects with an eye on their eventual implementation.

For example, different programming languages like assembler, PL/M, C, Pascal, and Fortran enjoy certain advantages over each other. Software developers should be able to draw on the most appropriate language to implement the different facets of a design. In order to support this kind of free choice, however, the development environment must be able to coordinate the use of a mix of programming languages, so that programmers can use different languages without concern about how the different modules will eventually be combined.

Like natural languages, the virtue of programming languages lies in their ability to represent abstract ideas in concrete terms. Just as it may be easier to express a certain idea with a particular natural language than another, programming languages vary in their ability to represent certain design concepts. For example, software engineers find that Pascal represents structured designs more faithfully than a language like Fortran. Also, languages like PL/M or C, which closely reflect the hardware base of a design, or assembly language, which provides the ultimate visibility into the hardware, are powerful tools for developing real-time embedded systems.

Still, programming languages share another feature with natural languages—varying degrees of popularity. For example, Fortran remains one of the most popular programming languages. Its continued strong momentum translates into a large installed base of

231257-4

software. For managers, this large installed base provides a ready source of existing code. On the other hand, managers must remain ready to incorporate newer languages like ADA into designs without starting from scratch.

In many software development projects, managers often look for a way to juggle several programming languages simultaneously. Software engineers can usually adapt quickly to new programming languages —particularly when they are supported by project management tools. On the other hand, the development environment often acts as a bottleneck in mixing several different languages in the same target system because of its inability to match the varying program and system interfaces of different languages.

The Intel development environment integrates different languages through a common object module format (OMF). A standard OMF works at several levels. During link time, OMF presents a standard method for indicating data type information, which the linker uses to build its memory allocation tables. Furthermore, debuggers exploit OMF's standard arrangement of symbolic information for handling symbolic debugging.

Two other aspects of the standard development environment include the definition of standard conventions for passing parameters between different programs—regardless of their implementation language —and standard interfaces to the operating environment. Besides accounting for critical implementation details another key measure of the effectiveness of a development environment is its support of application level standards like IEEE 754 for floating point operations or IEEE 802 for Ethernet.

For those areas currently without standards, the development environment takes the initiative with a baseline for the operating environment. Here, Intel's universal development interface (UDI) defines a system-independent interface between application programs and the operating environment. Rather than write their programs with system-dependent calls to operating system utilities, software developers use the same UDI call to allocate memory, for example, regardless of the target operating system. During link-time, the linker uses this UDI call to link in the appropriate system utility in iRMX™, for example (Fig 3). Consequently, programs that use the UDI can be ported between ISIS, iRMX, and Microsoft's Xenix simply by loading the modules into the new environment. Thus, if the design calls for a realtime operating environment like iRMX, engineers can develop the application under ISIS without fear that their work will be lost when the system is transported to the iRMX environment.

For the manager trying to improve productivity, no faster method exists than simply porting existing code to a new environment. Besides IEEE standards, which

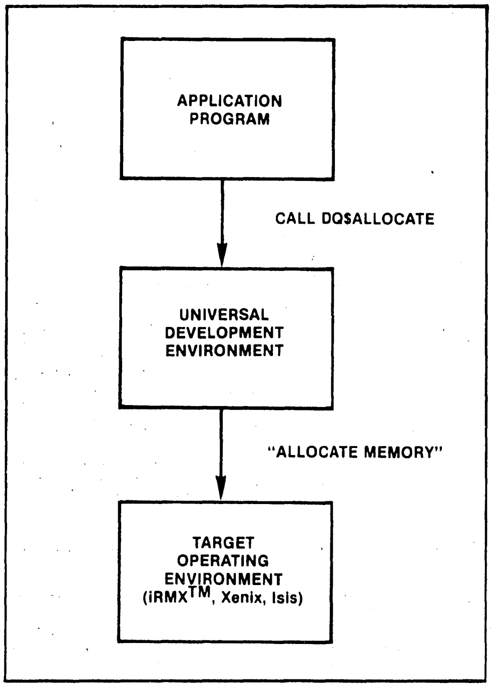

APPLICATION PROGRAM
CALL DQ$ALLOCATE
UNIVERSAL DEVELOPMENT ENVIRONMENT
"ALLOCATE MEMORY"
TARGET OPERATING ENVIRONMENT (iRMX™, Xenix, Isis)

Fig 3. Where applications standards do not already exist, a development system should follow some baseline. The universal development interface (UDI) sets a baseline for interactions between application programs and operating software. For example, an application that requires memory uses a UDI call (DQ$ ALLOCATE) which is later translated into the appropriate call for target operating environment.

provide a common application environment, the use of a common object format and universal development interface provide a clear migration path between operating environments.

SAME INTERFACE

In the kind of cross-development environments commonly used for creating microprocessor-based products, engineers work most effectively if they are able to split debugging into two phases. In the first phase, debugging occurs in parallel for the target hardware system and for the software. Here, engineers use the host environment to debug the basic logic of the software system. Once they are satisfied both with the logic of the software and with the operation of the hardware, the engineers then load the software into the target system for the second phase—integration and test.

This in-target phase is the critical step where hardware and software are finally integrated as a total system. As noted earlier, differences between the host and target environments can more than double costs. Consequently, a key feature of an integrated environment is a common debug interface between host and target.

Intel's PSCOPE debugger permits programmers to check out programs at the source-level both during logic debug and during in-target test. Because PSCOPE shows up again as one of the three major components of the I^{2}ICE system, software engineers are assured of a smooth transition between host and target. Along with PSCOPE, I^{2}ICE's in-circuit emulation and logic timing analyzer (LTA) give developers a full view simultaneously into the hardware and software components of their systems. Without this kind of coordinated approach to system integration and test, developers can never deal with the hardware and software as an integrated system, but are forced to switch continually between hardware testing and software debugging.

Supporting system integration at the most fundamental level, in-circuit emulation provides a transparent, full speed emulation of the iAPX 86 and iAPX 286 families of processors. Besides handling multiple level breakpoints and traces in single microprocessors, I^{2}ICE extends its support to multiprocessor environments. Developers can emulate a system of up to four microprocessors and examine complex processor interactions like synchronization. For example, I^{2}ICE lets engineers define events like breaks and traces conditionally, so that a microprocessor will break when another defined event occurs in a different microprocesor.

While I^{2}ICE and PSCOPE provide the fundamental support for a system's underlying hardware and software, the LTA also serves as a key element of the system's integrated package. Displaying 16 channels of

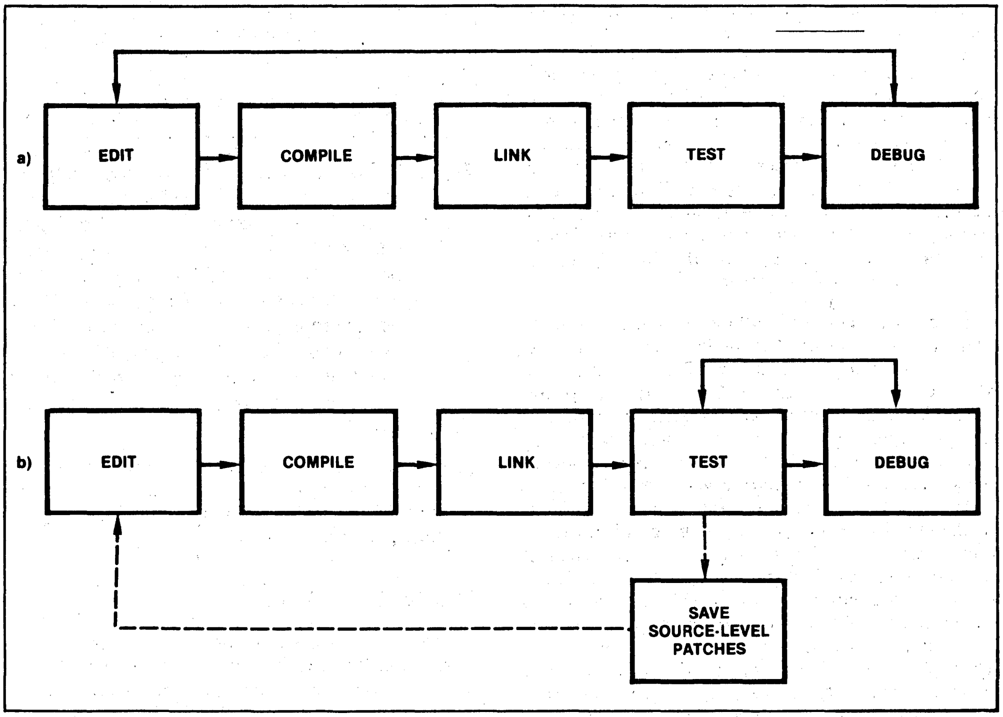


Fig 4 a/b. In the past, engineers have needed to iterate through a lengthy development cycle in order to debug source code in the target system (a). On the other hand, PSCOPE lets engineers use source level code to debug and patch target systems and continue debugging, then finally, after many bugs are found, save the source-level patches on disk for later addition to the original source files (b).

231257-6

logic and timing information, the LTA helps isolate critical state and timing problems. In order to speed the analysis process, this menu-oriented system also permits engineers to save debugging setups and waveforms on disk.

A key advantage of an integrated environment is its ability to present information, through a consistent command language, in a familiar form. With I^{2}ICE, this feature extends to logic and timing analysis. Rather than present a morass of digits, the LTA displays most information in easy to understand waveform diagrams.

Just as the LTA has moved system integration and test above the bit level, PSCOPE shortens software debugging by permitting engineers to test programs using their own symbols, rather than machine code. With the traditional machine code debugger, if they wanted to patch a section of machine code, programmers would spend hours converting machine code between different formats, like binary and hex, and calculating the machine code equivalents of assembler instructions. Even somewhat more sophisticated debuggers that disassemble machine code are little help in retaining the sense of a program as expressed through its use of symbols.

Instead, even though it helps software engineers deal with machine code when necessary, PSCOPE can handle debugging at the level of the original source code. Consequently, programmers can set an unlimited number of breakpoints by statement number, step through a single source statement at a time, and trace execution by statement number, procedure name, or label (regardless of whether they are working with the host or target system).

From the user's point of view, the utility of PSCOPE lies in its built-in, CRT-oriented editor and in its command languge that resembles a high level structured programming language (see the Table). Using PSCOPE's editor, engineers write extensive procedures in the command language for testing code and even patch existing code with new or revised source statements.

PSCOPE's ability to handle source-level patches avoids the conventional development scenario where software developers go through a continual cycle of edit-compile-link-test-debug [Fig 4(a)]. Source-level patching short-circuits this loop; programmers can remain in the debug phase—patching at the source-level and even saving the source-level patch on disk for later incorporation into the original source-code files maintained under SVCS [Fig 4(b)].

The advantages of an integrated environment show up here very dramatically. During compilation, the compiler places symbolic information associated with a program into the object modules it generates. In turn, the linker carries this information along into the run time image. Both PSCOPE and I^{2}ICE draw on this symbolic information for their source-level debugging. Consequently, during system debugging, developers see familiar procedure and data names, rather than a confusing series of machine codes or disassembled mnemonics. Furthermore, because it maintains this symbolic information in a virtual table, PSCOPE is able to handle arbitrarily long symbol tables—it just brings a new page of symbols from disk, if necessary.

As a result of its ability to coordinate its tools for the various stages of development, the Intel development environment lets system engineers concentrate on product development, rather than administrative chores. For the development manager, this translates into on-time product delivery, without the costs of additional resources.

231257-7

In-Circuit Emulators 3

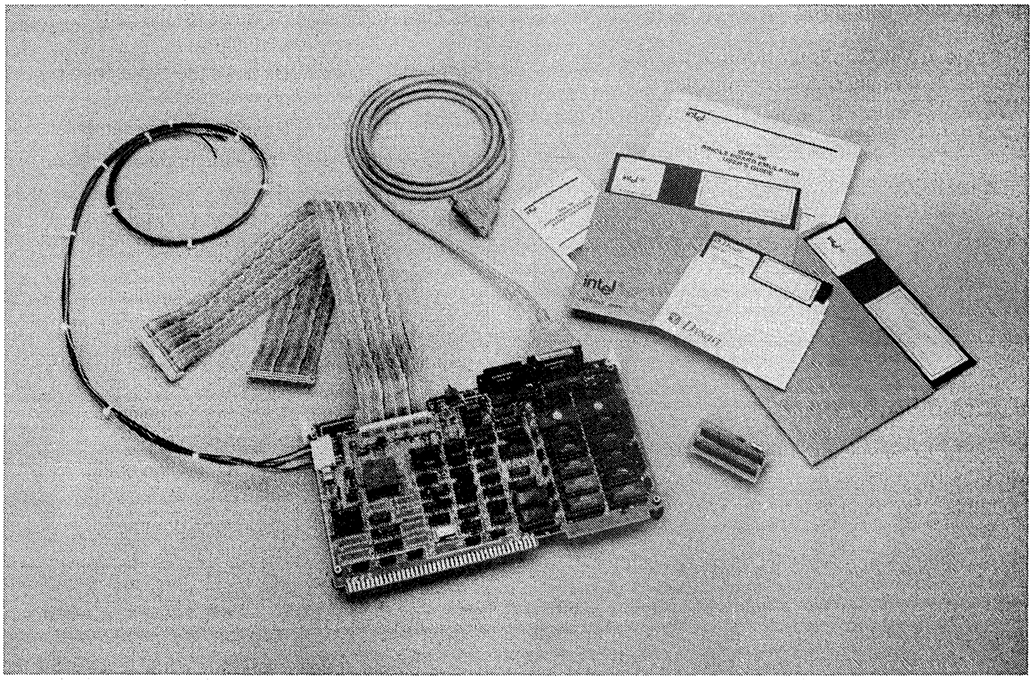
intel®

iSBE-96 DEVELOPMENT KIT SINGLE BOARD EMULATOR AND ASSEMBLER FOR THE MCS®-96 FAMILY OF MICROCONTROLLERS

- **Hosts**
  - **—Intellec® Series III/IV Development Systems**
  - **—IBM* PC AT, PC XT, and Compatibles (3.0)**
- **Eight Software Execution Breakpoints That Can Selectively Be Turned On and Off**
- **12 MHz Emulation Speed**
- **Single Line Assembler/Disassembler**
- **MCS®-96 Software Support Package**
- **Configurable Serial I/O**
- **17.75 of On-Board User Memory**
- **Optionally Expandable to 64K of On-Board User Memory**

The iSBE-96 emulator supports the execution and debugging of programs for the MCS-96 family of microcontrollers at speeds up to 12 MHz. The MCS-96 family configurations are shown in Table 1. The iSBE-96 emulator consists of an 8097 microcontroller, a serial port and cables, and an EPROM-based monitor that controls emulator operation and the user interface.

The iSBE-96 emulator is a combination of hardware and software that permits programs written for the MCS-96 family of microcontrollers to be run and debugged in the emulator's artificial environment or in the user's prototype system. As a result, development time can be reduced by the early integration of hardware and software.



231015-1

October 1986
Order Number: 231015-005

FUNCTIONAL DESCRIPTION

Integrated Hardware and Software Development

The iSBE-96 emualtor allows hardware and software development to proceed simultaneously. This approach is more time- and cost-effective than the alternate method: independent hardware and software development followed by system integration. With the iSBE-96 emulator, prototype hardware can be added to the system as it is designed; software and hardware integration occurs while the product is being developed. The emulator aids in the recognition of hardware and software problems.

Emulation is the controlled execution of the prototype software in the prototype hardware or in an artificial hardware environment that duplicates the microcontroller of the prototype system. The iSBE-96 emulator permits reading and writing of system memory, and control of program execution. The emulator also allows interactive debugging of the prototype software and can externally control program execution while operating in the prototype system. When the prototype system memory is not yet available, the iSBE-96 emulator's on-board memory permits software debugging.

Table 1. The Configurations of the MCS®-96 Family of Microcontrollers

		68 Pin	48 Pin
Digital I/O	ROMLESS	8096	8094
	ROM	8396	8394
	EPROM	8796	8794
Analog and Digital I/O	ROMLESS	8097	8095
	ROM	8397	8395
	EPROM	8797	8795

iSBE-96 Software

The iSBE-96 emulator software is available for use with the following host systems:

- Intellec Series III and Series IV development systems
- IBM PC/AT and PC/XT computer systems

The iSBE-96 emulator software is also available from U S Software* for use on the Intel Personal Development System (iPDS™) and the Intellec Series II development system.

***NOTE:**
U S Software is a registered trademark of United States Software Corporation.

The iSBE-96 emulator is supplied with a driver routine that communicates with the monitor software on the iSBE-96 emulator board through serial channel 1 or 2 (com1/com2). The driver interrupts the 8097 using the non-maskable interrupt (NMI) line for incoming keyboard input. The commands associated with the driver and the monitor are described in the following sections.

iSBE-96 Driver

iSBE-96 emulator is shipped with driver software for use on the Series III/IV development systems and the IBM PC AT/XT running PC DOS, version 3.0 or greater. The driver software provides a few easy-to-use commands. These commands are described in Table 2. ASM/DASM available on DOS version only.

Table 2. iSBE-96 Driver Commands

Driver Command	Function
ASM	Loads memory with MCS-96 assembly mnemonics.
DASM	Displays memory as MCS-96 assembly mnemonics.
EXIT	Exits the driver and returns to the host operating system.
<CONTROL> C	Same as for the EXIT command.
HELP	Displays the syntax of all commands.
INCLUDE	Specifies a command file.
<CONTROL> I	Turns the command file on and off.
<TAB>	Same as <CONTROL> I (turns the command file on and off).
LIST	Specifies a list file.
<CONTROL> L	Turns list file on and off.
<CONTROL> S	Stops scrolling of the screen display.
<CONTROL> Q	Resumes scrolling of the screen display.
<CONTROL> X	Deletes the line being entered.
<ESCAPE>	Aborts the command executing.

iSBE-96 MONITOR

The iSBE-96 monitor performs the following functions:

- Loads and saves user programs.
- Independently emulates user programs.

Table 3. iSBD Monitor Commands

Monitor Command	Function
BAUD	Sets up the baud rate.
BR	Permits display and setting of up to eight software breakpoints.
BYTE	Permits display and changing of a single byte or range of bytes of memory or a single byte of the 8097 internal registers.
CHANGE	Permits display and changing of a series of memory words or bytes.
<CONTROL> S	Stops scrolling of the screen display.
<CONTROL> Q	Resumes scrolling of the screen display.
<CONTROL> X	Deletes the line being entered.
<ESCAPE>	Aborts the command executing.
GO	Begins emulation and continues until an enabled breakpoint is reached or the escape key is pressed.
LOAD	Loads programs and data from disks.
MAP	Permits mapping of several preprogrammed memory maps; also permits configurable serial I/O and selective servicing of the watchdog timer.
PC	Displays and changes the program counter.
PSW	Displays and changes the program status word.
RESET CHIP	Resets the 8096 to power-up conditions.
SAVE	Saves programs and data to disks.
SP	Displays and changes the stack pointer.
STEP	Provides single-step emulation with selective display formats.
VERSION	Displays the monitor version number.
WORD	Permits display and changing of a single word or range of words of memory or a single word of the 8097 internal registers.

- Examines and changes memory contents.
- Examines registers.
- Maps the file capabilities of the serial ports (DS/DT).
- Maps different memory configurations.

The monitor commands are described in Table 3.

Integrating Hardware and Software

When the prototype system hardware is developed, the iSBE-96 emulator interfaces to the prototype through two 50-pin ribbon cables. The emulator can then execute code from the iSBE-96 on-board RAM (or from user-provided memory) and exercise the prototype system hardware.

BLOCK DIAGRAM

Figure 1 is a block diagram showing the iSBE-96 emulator. The following sections describe each block.

The Processor

The 68-pin processor of the iSBE-96 emulator is used only in the 8097 external-access mode. An 8097BH will be supported in 16-bit bus mode only.

An adapter board is provided for the 68-pin PGA version of the 8096 and 8097 microcontrollers. When debugging a 68-pin package, the two 50-pin ribbon cables plug into the 68-pin adaptor board which is plugged into the 68-pin socket on the prototype system.

When debugging a 48-pin package, the two 50-pin cables plug into the 48-pin adaptor board, which is then plugged into a 48-pin socket in the prototype system. A 68-pin PLCC Adaptor may be ordered.

iSBE-96 Emulator I/O

The iSBE-96 emulator's memory-mapped I/O devices are used to manage the system. These I/O devices are mapped into memory between locations 01F00H and 01FFFH.

Included as part of the I/O are two serial ports. One is configured as data set (DS) and the other as data terminal (DT). When operating with an Intellec® development system, the data set port is used as the system console and the link for exchanging files.

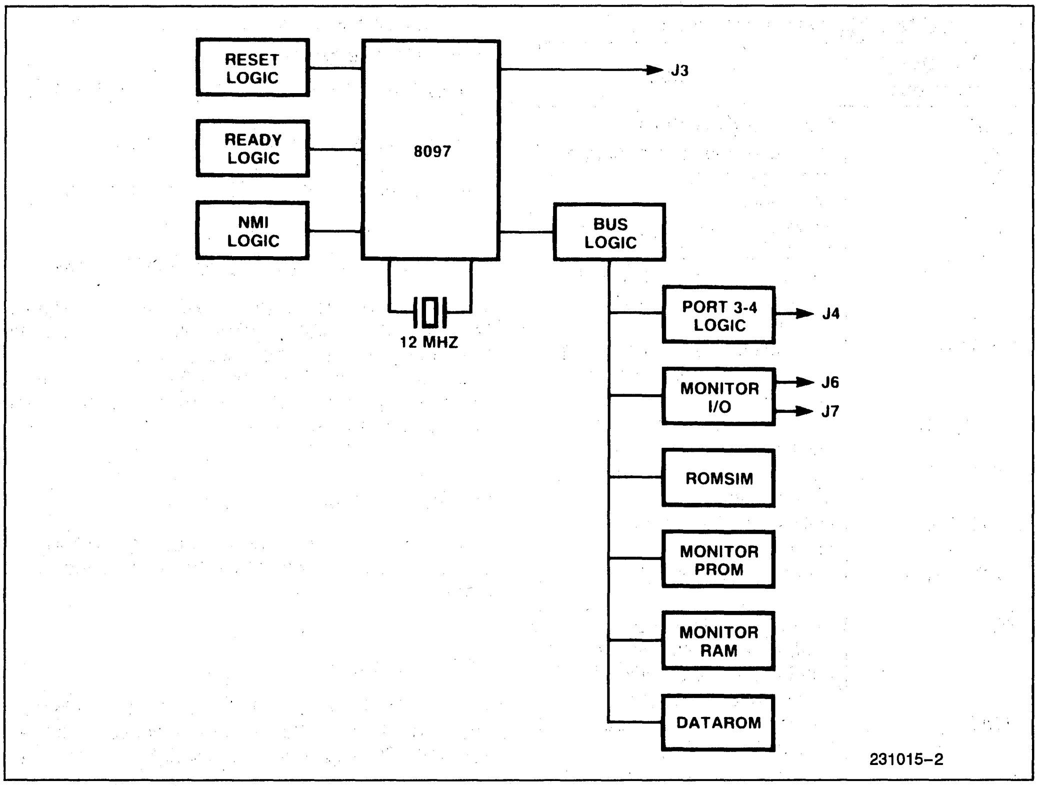


Figure 1. Block Diagram for the iSBE-96 Single Board Emulator

The serial ports are serviced under control of the NMI interrupt. The NMI interrupt has highest priority on the microcontroller and interrupts the user program when characters are entered from the keyboard. When in emulation, the monitor will still service inputs from the keyboard and execute certain monitor commands. Monitor activity is not totally transparent to the user.

Simulated ROM (ROMSIM)

There are eight 28-pin JEDEC byte-wide sockets with 2K-by-8 static RAMS present on the board. The partition on the user's prototype system that will be ROM is simulated by RAM on the iSBE-96 emulator board. This RAM facilitates easy program development, allowing users to correct and test problems in their programs.

ROMSIM can be expanded by replacing the iSBE-96 RAMs with 8K-by-8 static RAMs.

Port 3-4 Logic

The port 3-4 logic has two functions: to provide bus expansion and to provide I/O ports. The port 3-4 logic is controlled by a software switch available with the MAP command.

The iSBE-96 emulator reconstructs ports 3 and 4 of the 8394, 8395, 8396, and 8397 microcontrollers when the logic is defined by the MAP command as port 3-4. This port function should be selected when one of these four microcontrollers is intended as the target microcontroller.

When the BUS switch of the MAP command is specified, the iSBE-96 address/data expansion bus is available to the prototype system.

THE iSBE-96 EMULATOR MEMORY MAP

The target system should be designed with a memory map that is compatible with one of the iSBE-96

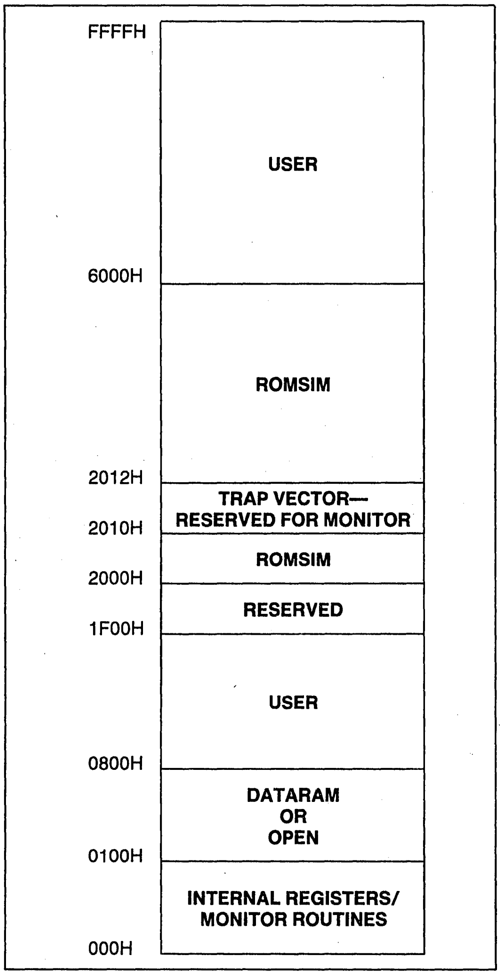


Figure 2. iSBE-96 Emulator Default Mapping

memory maps. Figure 2 shows the default address mapping. The following sections describe the areas of memory.

Internal Registers/Monitor Routines

Normally locations 000H through 0FFH contain the internal register space of the 8097. However, instruction fetches from these locations access external memory. This memory space contains the monitor's non-maskable interrupt service routine and utility routines.

For the monitor to access the user memory, the address and data is passed to the interrupt or utility routines. The routines then modify the mode register to enable user memory, disable all of the monitor's memory (except page zero), and perform the appropriate operation. After an operation is complete, the service and utility routines restore the mode register to its previous state and return to the main monitor code. The NMI service routine is used to handle the keyboard input on the serial port.

DATARAM

Locations 100H to 7FFH are mapped as the DATARAM space. This RAM is for general purpose use and is optionally enabled by using the MAP command. When the DATARAM buffer is not enabled, any access to this partition results in an access to user prototype system memory.

User Area

Locations 800H to 1EFFH are a user area. If an access is made to this partition, it is directed to the user's prototype system. Any memory mapped as I/O in the user system should be placed in this partition. With 8K-by-8 static RAMs, this area is located and available on the iSBE-96 board.

Reserved Area

Locations 1F00H to 1FFFH are reserved by the monitor for on-board I/O devices.

ROMSIM

Because some of the MCS-96 family of microcontrollers are ROMLESS parts, a user program can be loaded for execution into the on-board RAMS of the iSBE-96 emulator. Locations 2000H to 5FFFH are mapped to this RAM space; the space is called ROMSIM.

Trap Vector

Locations 2000H to 2010H are the interrupt vector locations. Vector address location 2010H is used by the iSBE-96 monitor for NMI.

User Area

The partition 6000H to 0FFFFH is mapped to the user prototype area. During emulation any access to this partition is directed to the user's prototype system.

EXPANDING ON-BOARD MEMORY

On-board memory can be expanded to a full 64K bytes by replacing the supplied 2K-by-8 static RAMs with 8K-by-8 static RAMs or PROMs. The user may also replace on-board ROMSIM memory with 2K-by-8 PROMs or even locate all 64K bytes of memory on the prototype system.

DESIGN CONSIDERATIONS

Designers should note the following considerations for designing with the iSBE-96 emulator:

- The iSBE-96 software uses 6 bytes of user stack space.
- Analog signal accuracy is impaired when driven over the emulator cable (up to ±50 mV loss of A/D conversion accuracy).
- The iSBE-96 emulator has some ac/dc parametric differences from the 8097 chip.
- The NMI vector is used for console service (Intel reserved interrupt).
- Keyboard activity during emulation affects real-time emulation because a 50 to 100 microsecond interrupt service routine is executed for every keystroke.
- The only hardware reset available for the iSBE-96 emulator is the system reset momentary switch (switch 1 on the emulator board).
- User system memory should be configured to the iSBE-96 memory map (see Figure 2).
- The iSBE-96 emulator does not support a user system crystal as shipped.
- The iSBE-96 driver software provided by Intel is not compatible with the Intellec Model 800 or Series II Development Systems.
- The IBM PC/AT and PC/XT have been evaluated and accepted by Intel as compatible hosts for its development systems. Intel has not evaluated any ohter PC DOS machines (3.0). However, Intel knows of no reason why these PC DOS machines would not be compatible hosts for its development systems.

SPECIFICATIONS

Equipment Supplied

Standard MULTIBUS®-size board assembly

EPROM-based monitor

Auxiliary power cable

RS-232 serial cables

Two standard, 18 in., 50-pin ribbon cables for connection to the user's prototype system

Adapter board for the 48-pin DIP and 68-pin PGA versions of the MCS-96 microcontroller

MCS-96 software support package

One 8 in. single-density software disk for the Series III

One 8 in. double-density software disk for the Series III

One 5¼ in. software disk for the Series IV

One 5¼ in. software disk for the IBM PC AT/XT

Documentation

ISBE-96 User's Guide (Order number 164116)
iSBE-96 Pocket Reference (Order number 164157)
Developing MCS-96 Applications Using iSBE-96 (Order Number 280249-001, AP-273)

Emulation Clock

12 MHz supplied crystal

Physcial Characteristics

Width: 6.75 in. (17.15 cm)
Length: 12 in. (30.48 cm)
Height: 0.75 in. (1.91 cm)

DC Electrical Requirements

Voltage	Current
+5V ± 5%	3.5a max
+12V ± 5%	0.06a max
−12V ± 5%	0.05a max

Environmental Characteristics

Operating Temperature:	10°C to 40°C
Operating Humidity:	10% to 85% relative humidity, without condensation

IBM PC XT/AT Host Requirements

- PC DOS, version 3.0 or greater
- External power supply
- Serial channel Com1

ORDERING INFORMATION

Intel 3065 Bowers Ave.
Santa Clara, CA 95051

Part Number	Description
SBE96SKIT	iSBE-96 single board emulator for use with the Series III/IV development systems. The kit contains the following parts: • iSBE-96 single board emulator • MCS-96 software support package for the Series III/IV development systems • iSBE-96 Series III/IV upgrade kit (cables and software needed to run on Intel Hosts)
SBE96DKIT	iSBE-96 single board emulator for use with the IBM PC/AT and PC/XT computer systems. The kit contains the following parts: • iSBE-96 single board emulator • MCS-96 software support package for PC DOS • iSBE-96 DOS upgrade kit (cables and software needed to run on the IBM PC/AT or PC/XT)
SBE96DU	iSBE-96 DOS upgrade kit for those customers who wish to upgrade their Series III/IV kit to run on the IBM PC AT or PC XT.
SBE96SU	iSBE-96 Series III/IV upgrade kit for those customers who wish to upgrade their DOS kit to run on Intel Hosts).
TASBEE	68-pin PLCC Adaptor Board.

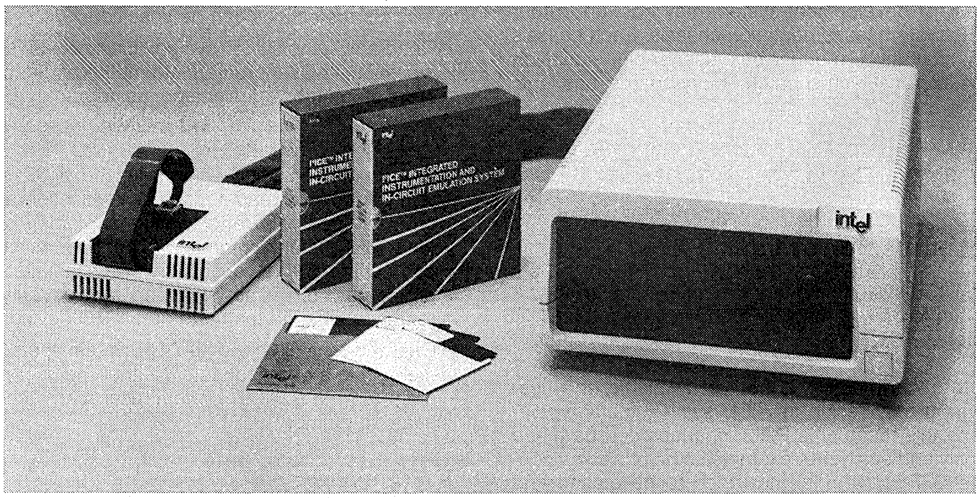
U S Software
5470 N. W. Innisbrook
Portland, OR 97229
Phone: 503-645-5043
International Telex 4993875

Part Number	Description
XASM96	Performs assembly of MCS®-96 programs written on the iPDS.
ATOP96	iPDS and Series II software for iSBE-96 host communications. Performs host communications and assembly/disassembly of iSBE-96 instructions. The XASM Host Cross Assembler software must be ordered with this software.

I²ICE™ Integrated Instrumentation and In-Circuit Emulation System

- Provides Real-Time In-Circuit Emulation
- Offers Symbolic Debugging Capabilities
  - Accesses Memory Locations and Program Variables (Including Dynamic Variables) Using Program-Defined Names
  - Maintains a Virtual Symbol Table
- Offers Multi-Condition, Multi-Level, Multi-Probe Break and Trace Capability
- Provides Built-In AEDIT Editor to Allow Editing of Development System Files without Exiting from I²ICE Operation
- Provides Low Cost Conversions Among 8086, 8088, 80186, 80188 and 80286 Microprocessors
- Simultaneously Controls up to Four Microprocessors for Debugging Multiprocessor Systems for a Single Work Station
- Supports Common Memory between Processor without Any User System Hardware
- Offers an Integrated 16-Channel 100-MHz Logic Timing Analyzer
- Maps User Program Memory into a Maximum of 288K Zero-Wait-State RAM (Zero Wait-States up to 10 MHz)
- Maps User I/O to Console or to Debugging Procedures
- Provides Disassembly and Single-Line Assembly to Help with On-Line Code Patching
- Common Human Interface Provided by the PSCOPE-86 Debugging Language and the I²ICE Command Language
- Uses Integrated Command Directory, ICD™, for Command Syntax Direction/Correction to Ease Debug Operations

The Intel Integrated Instrumentation and In-Circuit Emulation (I²ICE™) system aids the design of systems that use the 8086, 8088, 80186, 80188, and 80286 microprocessors. The I²ICE system combines symbolic software debugging, in-circuit emulation, and the optional Intel Logic Timing Analyzer (iLTA). Support features for the 8087 and 80287 coprocessors are also included. For the 8086/8088, 80186/80188, and 80286 processors, the I²ICE system support programs written in "C", PL/M, FORTRAN, Pascal, Ada*, and assembly language. Up to four I²ICE system instrumentation chassis can be hosted by one of Intel's Intellec® microcomputer development systems or by an IBM PC AT or PC XT.


I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM

210469–1

*Ada is a trademark of the Joint Ada Program Office. U.S. Department of Defense.

November 1986
Order Number: 210469-006

PHYSICAL DESCRIPTION

The I2ICE system hardware consists of the host interface board, the I2ICE system instrumentation chassis, the emulation base module, the emulation personality module, a host/chassis cable, inter-chassis cables (for multiple chassis systems), a user cable, optional high-speed memory boards, and an optional logic timing analyzer. The I2ICE system software consists of I2ICE system host software, I2ICE system probe software, confidence tests, PSOPE 86, and optional iLTA software. Table 1 shows elements of the I2ICE system.

The host interface board resides in the host development system. A cable connects the host interface board to the I2ICE system instrumentation chassis. Another cable connects the I2ICE system instrumentation chassis to the buffer box.

The instrumentation chassis contains high-speed zero-wait-state emulation memory, break-and-trace logic, memory and I/O maps, and the emulation clips assembly.

The chassis may also contain the optional logic timing analyzer and optional high-speed memory. High-speed memory is expandable from 32K bytes to 288K bytes in 128K increments.

The buffer box contains the emulation personality module. This module configures the I2ICE system for a particular iAPX microprocessor. The user cable connects the buffer box to user prototype hardware.

The host development system may host up to four I2ICE system instrumentation chassis. Each chassis may have its own buffer box, user cable, emulation clips, optional high-speed memory boards, and logic timing analyzer.

FUNCTIONAL DESCRIPTION

Resource Borrowing

The I2ICE system memory map allows the prototype system to borrow memory resources from the I2ICE system.

If prototype memory is not yet available, the user program may reside in I2ICE system memory. Because this memory is RAM, changes can be made quickly and easily. For example, if the prototype contains EPROM, it does not need to be erased and reprogrammed during development.

Later, as prototype memory becomes available, the verified user program can be reassigned, memory block by memory block to prototype memory.

The I2ICE™ System Memory Map

The I2ICE system can direct (map) an emulated microprocessor's memory space (the user program memory) to any combination of the following:

- High-speed I2ICE system memory—this consists of 32K bytes of programamble wait-state memory (programmable from 0 to 15). This memory resides in the I2ICE system chassis on the map-I/O board.
- Optional high-speed I2ICE system memory—this consists of up to 256K bytes of programmable wait-state memory (0 wait-states up to 10 MHz). This memory resides in the I2ICE system chassis on one or two optional high-speed memory boards (128 K bytes each).
- MULTIBUS® bus memory (host system memory)—this resides in the host development system itself. (Any amount of unused host memory can be used in 1K increments.) Note that this feature is not available for a PC host.
- User memory—this resides in the user prototype hardware.

When a user program runs in I2ICE system memory or user memory, the I2ICE system emulates in real time. A memory access to MULTIBUS bus memory, however, inserts approximately 25 wait-states into the memory cycle.

Access Restrictions

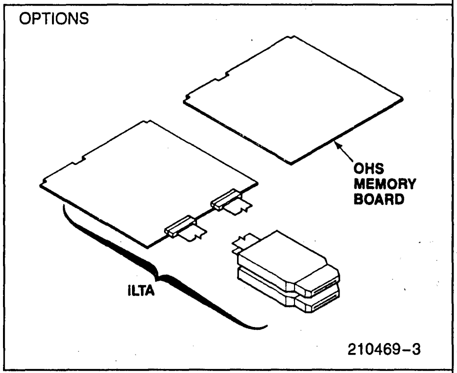
In addition to directing memory accesses, the following access restrictions can be specified:

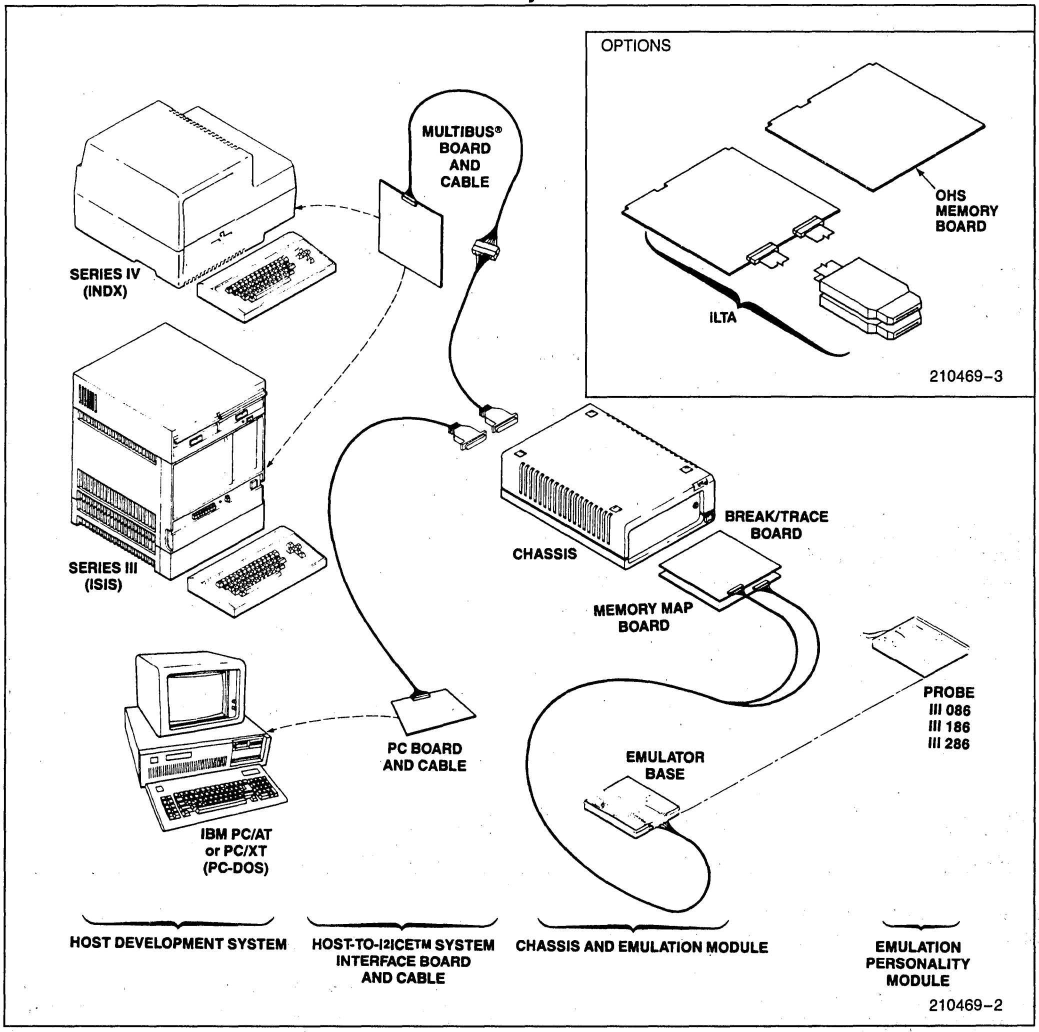
- Read-only—the I2ICE system displays an error message if a user program attempts to write to an area of memory designated as read-only. The user can, however, write to a read-only area with I2ICE system commands.
- Read/write, no verify—normally, the I2ICE system performs a read-after-write verification after program loads and after writing to memory with an I2ICE system command. The I2ICE system can suppress this verification. For example, if a prototype has memory-mapped I/O, a verifying read may change the state of the I/O device.
- Guarded—initially, the I2ICE system puts all memory in a guarded state. Neither the user program nor the I2ICE system user can access guarded memory.

The I2ICE™ System I/O Map

The I2ICE system can direct (map) an emulated microprocessor's I/O space to the host development

Table 1. I²ICE™ System Overview





Name	Description
Host Development System	Required for all applications. Use one of the following: • Intellec Series III development system • Intellec Series IV development system • IBM PC AT or PC XT (with 512K bytes of available memory and version 3.0 of PC DOS) • IBM 50 system (available in Japan; features *kanji*)

Table 1. I²ICE™ System Overview (Continued)

Name	Description
Host-to-I²ICE System Interface Board, Cable, and Host Software	Required for communication between the host and the I²ICE system. • MULTIBUS® bus interface board for Series III and Series IV (product code III520) • Host-to-I²ICE system cable for Series III and Series IV (product code III530 or III531) • I²ICE system host software for the Series III and Series IV (product code III951A, B, or C) • Package with PC host interface board, cable and PC DOS version of I²ICE host software (product code III520AT954D)
Instrumentation Chassis and Emulation Module	Required for real-time microprocessor emulation, break and trace capability, and memory and i/O capability. • Instrumentation chassis (product code III514B) has four board slots: 1 slot for break/trace board 1 slot for map-I/O board 2 slots for 1 (or 2) optional high-speed memory board(s) and/or 1 optional logic timing analyzer board • Maximum of four chassis for multi-probe applications • Emulation module (product code III620) includes break/trace board, map-I/O board, and buffer base box
Emulation Personality Module (Probe) and Probe Software	Required for emulation of specific microprocessors: 8086/8088, 80186/80188, or 80286. • Module includes personality board, buffer box cover, and user cable • Series III or IV: Order probe and probe software separately • PC host: Probe and probe software packaged together
Logic Timing Analyzer (iLTA) [not shown]	Required for acquisition and storage of events and glitches for signal measurement applications. • Complete with iLTA board (mounts in instrumentation chassis), probe pods, and cables • User Series III or Series IV host (cannot be used with the IBM PC AT and PC XT)
Optional High-Speed Memory Board (OHS) [not shown]	Required for memory expansion. • 128K bytes of programmable (0 to 15) wait-state memory • One or two boards mount in the instrumentation chassis

system's console, to the prototype system, to debugging procedures, or to a combination of these.

SIMULATING I/O WITH THE HOST DEVELOPMENT CONSOLE

Suppose a user program requires input from an I/O device not yet part of the prototype. Map the input port range assigned to that device to the host development systems' console. Then, when the user program requires input, it halts and the I²ICE system console displays a message requesting the data. When you enter the required data at the keyboard, the user program continues.

SIMULATING I/O WITH I²ICE™ SYSTEM DEBUGGING PROCEDURES

Procedures that supply the needed input data can be written in the I²ICE system command language. When setting up the I/O map, the user specifies that the I/O procedure is invoked when certain I/O ports are accessed.

I/O ports are mapped in blocks of 64 byte-wide ports or 32 word-wide ports. A total of 64K byte-wide ports or 32K word-wide ports can be mapped.

Symbolic Debugging

With symbolic debugging, a memory location can be referenced by specifying its symbolic reference. A symbolic reference is a procedure name, line number, or label in the user program that corresponds to a location in the user program's memory space.

TYPICAL SYMBOLIC FUNCTIONS

Symbolic functions include:

- Changing or inspecting the value and type of a program variable by using its program-defined name, rather than the address of the memory location where the variable and a hexadecimal value for the data are stored.
- Defining break and trace events using source-code symbols.

With symbolic debugging, the user can reference static variables, dynamic (stack-resident) variables, based variables, and record structures combining primitive data types. The primitive data types are ADDRESS, BOOLEAN, BYTE BCD, CHAR, WORD, DWORD, SELECTOR, POINTER, three INTEGER Types, and four REAL types.

THE VIRTUAL SYMBOL TABLE

The I²ICE system maintains a virtual symbol table for program symbols; that is, the entire symbol table need not fit into memory at the same time. (The size of the virtual symbol table is constrained only by the capacity of the storage device.)

The I²ICE system divides the symbol table into pages. If a program's symbol table is large, the I²ICE system reads only some of the symbol table pages into memory. When the user references a variable whose symbol is not currently defined in memory, the I²ICE system reads the needed symbol table page from disk into memory.

Breakpoint, Trace, and Arm Specifications

With I²ICE system commands, breakpoint, trace, and arm specifications can be defined.

Breakpoints allow halting of a user program in order to examine the effect of the program's execution on the prototype. With the I²ICE system, a breakpoint can be set at a particular memory location or at a particular statement in a user program (including high-level language programs). A break can also be set to occur when the user program enters or accesses a specified memory partition or reads or writes a user program variable. When the user program resumes execution, it picks up from where it left off.

Normally, the I²ICE system traces while the user program executes. With a trace specification, however, the user can choose to have tracing occur only when specific conditions are met.

An arm specification describes an event or combination of events that must occur before the I²ICE system can recognize certain breakpoint and trace specifications. Typical events are the execution of an instruction or the modification of a data value.

The I²ICE system command language allows you to specify complex, multilevel events. For example, you can specify that a break occurs when a variable is written, but only if that write occurs within a certain procedure. The execution of the procedure is the arm condition; the variable modification is the break condition. The I²ICE system command language allows users to specify complex events with up to four states with four conditions and to use such events as arm, break, or trace conditions; a specified number of events can be used as a condition.

Coprocessor Support

The 8086/8088 emulation personality module provides transparent RQ/GT and MN/MX pin emulation to support real-time prototype systems that use the 8087 as a coprocessor. The 8086/8088 (and the 80186/80188) emulation personality module also provides debugging features specific to the 8087. I²ICE system commands provide access to the 8087's stack, status registers, and flags. The I²ICE system's disassembly and trace features extend to 8087 instructions and data types.

The 80186 and 80286 emulation personality modules also allow the prototype hardware to contain coprocessors. The 80186 probe can qualify break points and collect trace information when the coprocessor drives the status lines ($\overline{S0}-\overline{S2}$) in the prescribed manner. The 80286 personality module allows the hardware to contain the 80287 processor extension and provides special debugging features—the user can enable and disable the 80287 and change and examine its registers.

DUBUGGING WITH THE I²ICE™ SYSTEM

The I²ICE system allows both hardware and software debugging (see Figure 1).

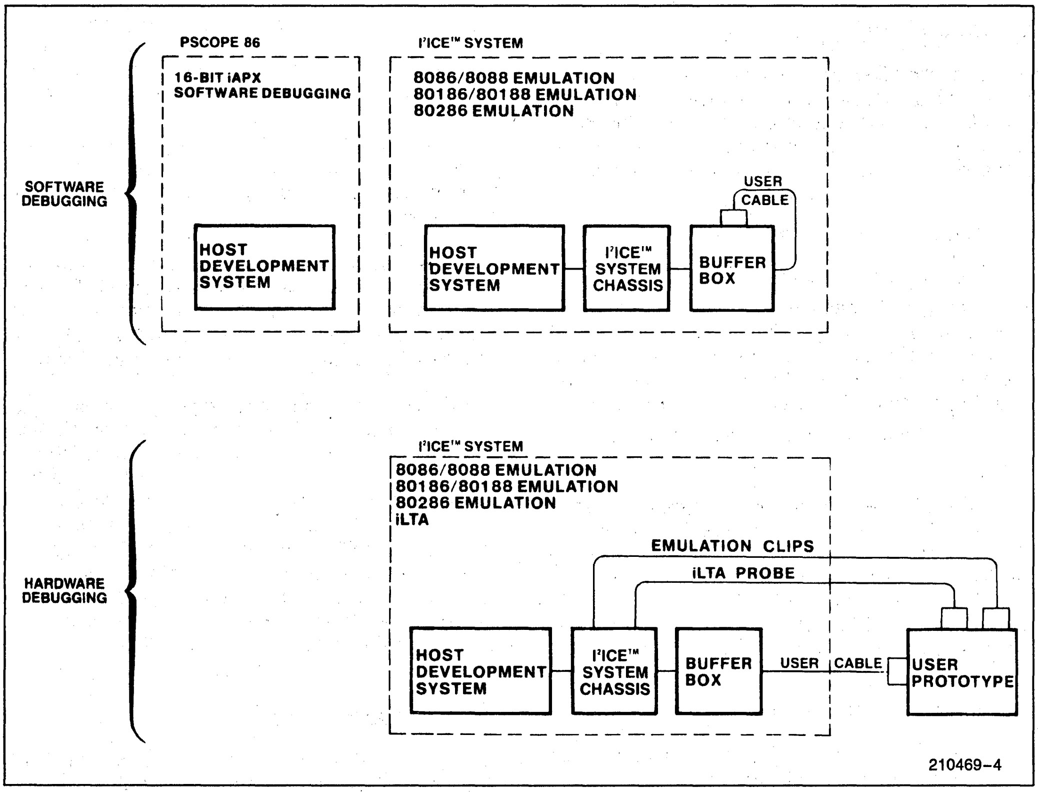


Figure 1. I²ICE™ System Debugging Capabilities

- Software debugging—I²ICE system commands permit symbolic debugging of user programs written in high-level languages as well as assembly language. By looping the user cable back into the buffer box, a user program can be debugged even if no prototype hardware is present. In a multi-probe environment, the I²ICE system can map common memory from the host development system and support semaphore operation even with no user system prototype hardware. This feature makes possible detailed debugging of multi-processor software before the hardware is available.
- Hardware debugging—the I²ICE system is a real-time, in-circuit emulator. Trace data are collected in real time, and I²ICE system software does not intrude into user program space. The optional iLTA adds the high-speed timing and data acquisition of a logic timing analyzer.

The userfulness of an I²ICE system extends throughout the development cycle, beginning with the symbolic debugging of prototype software and ending with the final integration of debugged software and prototype hardware.

PSCOPE 86

PSCOPE 86 is a high-level language, symbolic debugger, designed for use with Pascal 86, PL/M 86, and FORTRAN 86. It is a separate product included with Series III and Series IV versions of the I²ICE system; it runs in the host development system. PSCOPE 86 is field-proven, familiar to Intel customers, and suited for the debugging of applications software when the hardware capabilities of the I²ICE system are not needed. The PSCOPE 86 and I²ICE system command languages are similar. (Note that PSCOPE 86 is available as an option for use with the PC AT or PC XT.)

Designing a product that contains a microcomputer requires close coordination of hardware and software development. A typical design process takes advantage of both the I²ICE system and PSCOPE 86. Use PSCOPE 86 for debugging software before downloading the software into a target environment; use the I²ICE system for debugging and emulation in the target system.

THE I²ICE™ SYSTEM COMMAND LANGUAGE

The syntax of I²ICE system commands resembles that of a high-level language. The I²ICE system command langauge is versatile and powerful while remaining easy to learn and use.

The Integrated Command Directory (ICD™) assists users with command syntax.

- The ICD directory directs the user in choosing commands from display on the bottom line of the screen. As commands are entered, the bottom line indicates syntax elements available for use in the commands.
- The ICD directory flags syntax errors. Syntax errors are flagged as they occur (rather than after the carriage return is pressed).
- The ICD directory provides on-line help with the HELP command.

Automatic expansion of LITERALLY expressions is available. When the feature is activated, each character string defined by a LITERALLY definition is automatically expanded to its full length.

The I²ICE system command language deals with user-created, debugging objects. By manipulating debugging objects, the user can streamline complex debugging sessions.

Debugging objects are uniquely named, user-created, software constructs that the I²ICE system uses to manage the debugging environment. The four types of debugging objects are: debugging procedures, LITERALLY definitions, debugging registers, and debugging variables. In the following examples, I²ICE system keywords are shown in all caps.

- Debugging procedures (named groups of I²ICE system commands) can simulate missing software or hardware, collect debugging information, and make troubleshooting decisions. For example, consider a debugging procedure (called **init**) that simulates input from I/O ports 2 and 4.

  The procedure and MAPIO command are given first, followed by an explanation.

```
*DEFINE PROCEDURE init = DO
.*IF %0 = = 2 THEN
..*PORTDATA = 100T
..*ELSE IF %0 = = 4 THEN
...*PORTDATA = 65T ...*END
..*END
.*END
*MAPIO 0 LENGTH 64K ICE (init)
```

  Whenever the MAPIO command maps I/O ports to an I²ICE system procedure, three parameters are made available to the procedure (even if the procedure does not use them): %0, %1, %2. The parameter %0 passes the port number; %1 passes a Boolean value that indicates whether read or write I/O activity will occur; and %2 passes a Boolean value that indicates whether the I/O is a byte-wide or a word-wide port. PORTDATA is a pseudo-variable that contains the actual port data. This procedure specifies that if port 2 is used, the procedure returns 100 (base ten); if, however, port 4 is used, the procedure returns 65 (base ten).
- LITERALLY definitions are shorthand names for previously defined character strings. LITERALLY definitions can save keystrokes and improve clarity. For example, here is the definition of a LITERALLY that saves keystrokes. This LITERALLY allows the user to type DEF for DEFINE.

```
*DEFINE LITERALLY DEF = "DEFINE"
```

  These definitions may be saved to disk and autoreloaded. In addition, an automatic LITERALLY expansion feature can be turned on and off.
- Debugging registers are user-created, software registers that hold arm, breakpoint, and trace specifications. The I²ICE system can be ordered to emulate the user program and specify one or more debugging registers. There is no need to reenter the specificatoin for each emulation. For example here is the definition of a debugging register called **pay** that contains a trace specification. This example takes advantage of the previous LITERALLY definition.

```
*DEF TRCREG pay = :cmaker.payment
```

To emulate a user program and trace only during the procedure **payment,** specify the debugging register **pay** as part of the GO command.

```
*GO USING pay
```

- Debugging variables are user-created variables used with I^{2}ICE system commands. For example, here is the definition of a debugging variable called **begin.** Its type is POINTER.

  ```
  *DEFINE POINTER begin = 0020H:0006H
  ```

  During a debugging session, the user can set the execution point to this pointer value by typing:

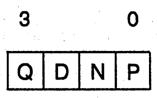
  ```
  *$ = begin
  ```

  The I^{2}ICE system pseudo-variable $ represents the current execution point.

Example of a Debugging Session

Figures 2, 3, and 4 illustrate some of the key capabilities of the I^{2}ICE system. The user program is written in Pascal-86. It was compiled, linked, and located on an Intellec Series III development system. The resulting file consists of absolute code and is called CMAKER.86. Figure 2 shows the Pascal listing; Figure 3 shows a sample debugging session; and Figure 4 briefly explains the debugging steps shown in Figure 3.

The CMAKER.86 program controls an automatic changemaker. The program reads the amount tendered (the variable **paid**) and the amount of the purchase (the variable **purchase**). It calculates the coins needed for change and asserts control signals to a change release mechanism by writing an output port. Each of the lower four bits of the output port controls the release of a different coin denomination.



Q = quarters
D = dimes
N = nickels
P = pennnies

I^{2}ICE™ System Command Functions

The I^{2}ICE system command language contains a number of functional categories.

- Emulation commands—the GO command instructs the I^{2}ICE system to begin emulation. The user can also command the I^{2}ICE system to break or trace under certain specified conditions.
- Utility commands—these are general purpose commands for use in a debugging environment. For example, one use of the EVAL command is to calculate the nearest source-code line number that corresponds to the address of an assembly language instruction. The HELP command provides on-line assistance. The EDIT command invokes a menu-driven text editor (AEDIT) that allows updating of debugging object definitions and editing of development system files without exiting from the I^{2}ICE system. A command line editior and history key are also provided.
- Environment commands—these are commands that set up the debugging environment. For example, the MAP command sets up the memory map. Another environment command (WAIT-STATE) inserts wait-states into memory accesses, allowing the simulation of slow memories.
- File handling commands—these are commands that access disk files. Debugging object definitions can be saved in a disk file and loaded in later debugging sessions. Debugging sessions can also be recorded in a disk file for later analysis.
- Probe-specific commands—these are commands whose effects are different for different probes. For example, the PINS command displays the state of selected signals lines on the current probe.
- Option-specific commands—these are commands that control an optional test/measurement device, such as the logic timing analyzer.

I^{2}ICE™ SYSTEM INSTRUMENTATION SUPPORT

I^{2}ICE™ System Emulation Clips

Eight external input lines are sampled during each processor bus cycle. The I^{2}ICE system records the values of these lines in it trace buffer during each execution cycle. The I^{2}ICE system can use these values when defining events.

Four additional output lines synchronize I^{2}ICE system events with external hardware. Two lines are active and programmable with I^{2}ICE system commands. Two other lines, break and trace, allow an I^{2}ICE system chassis to be linked to other I^{2}ICE system chassis.

Intel Logic Timing Analyzer (iLTA)

The iLTA analyzer is a chassis-resident, test/measurement module designed to extend the capability of the I^{2}ICE system to recognize events and collect data. The iLTA and the I^{2}ICE system emulator work together. They can trigger and arm/disarm each other. In addition, waveforms acquired by the

```
SERIES-III Pascal-86, V2.0
Source File: CMAKER.SRC
Object File: CMAKER.OBJ
Controls Specified: XREF, DEBUG, TYPE

STMT  LINE  NESTING     SOURCE TEXT: MAKER.SRC
   1     1   0  0       PROGRAM cmaker;
   2     2   0  0       VAR change,coins                                :integer;
   3     3   0  0           quarters,nickels,dimes,pennies              :integer;
   4     4   0  0           paid,purchase                               :word;

   5     6   0  0       PROCEDURE payment;
   6     7   1  0           VAR numberofcoins                           :integer;
   7     8   1  0               release                                 :word;
   8     9   1  0           BEGIN    (*payment*)
   8    10   1  1           numberofcoins:=quarters+dimes+nickels+pennies;
   9    11   1  1               while numberofcoins<>0 do
  10    12   1  1               BEGIN
  10    13   1  2               release:=0;
  11    14   1  2               if quarters<>0 then
  12    15   1  2                   BEGIN
  12    16   1  3                   release:=release+8;
  13    17   1  3                   quarters:=quarters-1
                                    END;

  15    19   1  2               if dimes<>0 then
  16    20   1  2                   BEGIN
  16    21   1  3                   release:=release+4;
  17    22   1  3                   dimes:=dimes-1
                                    END;
  19    24   1  2               if nickels<>0 then
  20    25   1  2                   BEGIN
  20    26   1  3                   release:=release+2;
  21    27   1  3                   nickels:=nickels-1
                                    END;
  23    29   1  2               if pennies<>0 then
  24    30   1  2                   BEGIN
  24    31   1  3                   release:=release+1;
  25    32   1  3                   pennies:=pennies-1
                                    END;
  27    34   1  2               numberofcoins:=quarters+dimes+nickels+pennies;
  28    35   1  2               OUTWRD(130,release);
  29    36   1  2               END;
  31    37   1  1           END;     (*payment*)

  32    39   0  0       BEGIN   (*main*)
  32    40   0  1       INWRD(2,paid);
  33    41   0  1       INWRD(70,purchase);
  34    42   0  1       change   :=paid-purchase;
  35    43   0  1       coins    :=change mod 100;
  36    44   0  1       quarters: =coins div 25;
  37    45   0  1       coins    :=coins mod 25;
  38    46   0  1       dimes    :=coins div 10;
  39    47   0  1       coins    :=coins mod 10;
  40    48   0  1       nickels  :=coins div 5;
  41    49   0  1       pennies  :=coins mod 5;
  42    50   0  1       payment;
  43    51   0  1       END.    (*main*)
```

210469-5

Figure 2. Listing of CMAKER.86

```
(1)  *BASE
     DECIMAL
(2)  *MAP 0K LENGTH 32K HS
     *MAPIO 0T LENGTH 192T ICE
     *MAP
     MAP          0K LENGTH      32K HS
     MAP         32K LENGTH     992K GUARDED
     *MAPIO
     MAPIO 00000H LENGTH        000C0H ICE
     MAPIO 000C0H LENGTH        0FF40H USER
(3)  *LOAD :F1:CMAKER.86
(4)  *DEFINE POINTER begin = $
     *DEFINE BRKREG pay = :cmaker #9
     *DEFINE PROC display = DO
     .*WRITE USING (' "quarters = ",T,0,>')quarters
     .*WRITE USING (' "dimes = ",T,0')dimes
     .*WRITE USING (' "nickels = ",T,0,>')nickels
     .*WRITE USING (' "pennies = ",T,0')pennies
     .*RETURN TRUE
     .*END
(5)  *GO USING pay
     ?UNIT 0 PORT 2H REQUESTS WORD INPUT (ENTER VALUE)*100
     ?UNIT 0 PORT 46H REQUESTS WORD INPUT (ENTER VALUE)*65
     Probe 0 stopped at :CMAKER #9 + 4 because of execute break
       Break register is PAY Trace Buffer Overflow
(6)  *quarters;dimes;numberofcoins
     +1
     +1
     +2
(7)  *DEFINE SYSREG wr_number = WRITE AT .:cmaker.payment.numberofcoins &
     **CALL display
     *GO USING wr_number
       quarters = + 1    dimes = +1
       nickels = +0      pennies = +0
     Probe 0 stopped at :CMAKER # 28 + 3 because of bus break
       Break register is WR_NUMBER
(8)  *numberofcoins
     +0
     *EVAL release
     1100Y 12T CH'..'
(9)  *CLIPSOUT = 11Y
(10) *GO FOREVER
     ?UNIT 0 PORT 82H OUTPUT WORD 0C
     ?Probe 0 stopped at location 0033:00AEH because of bus not active
       Bus address = 0203DE
     *$ = begin
     *
```

Figure 3. Sample Debugging Session (Explanations in Figure 4)

(1) Checking to see that the default radix is decimal.

(2) Mapping user program memory to I²ICE high-speed memory and user I/O ports to the I²ICE system console.

(3) Loading the user program.

(4) Defining debugging objects.

The debugging variable **begin** is set to $, an I²ICE pseudo-variable representing the current execution point. At this point is the debugging session, $ is the beginning of the user program.

The break register **pay** specifies a breakpoint at statement 9 in the user program.

The debuggning procedure **display** displays the value of some user program variables on the console.

(5) Beginning emulation with the debugging register **pay.** The console requests the two input values, **paid** and **purchase.** Then, the break occurs.

(6) Displaying three user program variables.

(7) Defining another debugging register. The specified event is the writing of the user program variable **numberofcoins.** When that event occurs, the I²ICE system calls the debugging procedure **display.** In addition to displaying some user program variables, this debugging procedure returns a Boolean value. Because this value is TRUE, the break occurs; if the value were FALSE, emulation would continue.

(8) Displaying the two user program variables, **numberofcoins** and **release.** The EVAL command displays **release** in binary, decimal, hexadecimal, and ASCII. Unprintable ASCII characters appear as periods (.).

(9) Asserting both output lines on the emulation clips. These lines are input to the prototype hardware and control a change release mechanism.

(10) Resuming emulation. The console displays the write of **release** to the output port. The user program finishes exeucting, and the probe stops emulating because of bus inactivity. The $ is set back to the beginning of the user program in preparation for another emulation.

Figure 4. Explanation of Sample Debugging Session in Figure 3

iLTA can be time-aligned with I²ICE system traces. (Note that iLTA is not available for use with the PC AT or PC XT.)

The iLTA analyzer brings the flexibility of high-speed triggering and glitch detection to the I²ICE system. The iLTA is a general purpose logic timing analyzer, supplemented with special features for microsystem debugging and I²ICE system integration. Following are some of iLTA's features.

- 16-channel, 100 MHz asynchronous operation
- 16-channel, 50 MHz asynchronous operation
- Single- or double-height timing waveforms presented with data scrolling, magnification, and delta-time read-out features.
- Minimum 3 nanosecond glitch detection (3 ns + 1 ns/volt for signal swings greater than 3 volts)
- A dual-threshold acquisition mode, with programmable logic level thresholds.
- A burst acquisition mode with window boundary indicators.
- User-defined channel labels and state display radixes.
- Disk storage for preservation and restoration of analyzer setups and acquired waveforms.
- Logic waveform comparison features (compares current acquisitions with pervious traces stored in auxiliary memory or on disk).

- Menu-driven operation and user-friendly display. The display takes advantage of screen highlighting, blinking characters, and reverse video.
- Powerful post-processing data analysis commands that are part of the I²ICE system command language.
- Multiple emulator break/trace and iLTA trigger/trace conditions may be shared with as many as four emulators and four iLTAs.

I²ICE™ SYSTEM SPECIFICATIONS

Host Requirements

Series III, Series IV, Model 800, or IBM PC AT or PC XT.

512K bytes in host development system memory space.

Two double-density diskette drives or a hard disk.

For the iLTA to run on a Series III, the III-820 board must be installed. Model 800 systems and the IBM PC AT and PC XT systems do not support the iLTA option.

I²ICE™ System Software

I²ICE system host software
I²ICE system probe software
I²ICE system confidence tests
I²ICE tutorial
PSCOPE 86 (not currently available for PC-DOS)
Optional iLTA software and iLTA confidence tests (not available for PC-DOS)

System Performance

Mappable zero wait-state memory (zero wait-states up to 10 MHz for 8086; 8 MHz for 8088 and 80186/80188; and 6 or 8 MHz for 80286):	Minimum 32K bytes, maximum 288K bytes
Trace buffer:	1023 x 48 bits
Virtual symbol table:	The number of user program symbols is limited only by available disk space

Physical Characteristics

INSTRUMENTATION CHASSIS

Width:	17.0 in (43.2 cm)
Height:	8.25 in (21.0 cm)
Depth:	24.13 in (61.3 cm)
Weight:	48 lbs (21.9 kg)

HOST/CHASSIS CABLE

10 ft (3.0m) and 40 ft (12.2 m) options for Series III/Series IV host
15 ft (4.6m) for PC host

INTER-CHASSIS CABLE SET

2 ft (61 cm) and 10 ft (3.0m) options

BUFFER BOX

Width:	8.5 in (21.6 cm)
Height:	3.0 in (7.6 cm)
Depth:	10.0 in (25.4 cm)
Weight:	8 lbs (3.7 kg)

Electrical Characteristics

90–132V or 180–264V (selectable)
47–63 Hz
12 amps (AC)

Environmental Requirements

Operating Temperature:	0°C to 40°C (32°F to 104°F)
Operating Humidity:	Maximum of 85% relative humidity, non-condensing

Emulation Clips

Emulation clipsin lines are sampled once every bus cycle when the address bits become valid on the address bus. During emulation, the I²ICE system records the value of the clipsin lines in the trace buffer once very execution cycle.

Table 2. I²ICE™ Emulation Clips—DC Characteristics

Signal	Input Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low I_{IL} μA	High I_{IH} μA	Low I_{OL} mA	High OH mA
Clipsout Lines					33 at 0.7V	4.8 at 2.0V
SYSBREAK SYSTRACE					38 at 0.7V	1.0 at 2.0V
Clipsin Lines	1.05	2.5	50	50		

I^2ICE™ SYSTEM 8086/8088 PROBE HIGHLIGHTS

- Provides up to 10 MHz real-time emulation
- One-megabyte addressing
- Emulates both Minimum and Maximum modes
- Provides 8087 coprocessor support

Table 3. I^2ICE™ System 8086/8088 User Interface—DC Characteristics

Signal	Input Voltage		Output Voltage		Input Current		Output Current			
	Max V_{IL} V	Min V_{IH} V	Max V_{OL} V	Min V_{OH} V	Max I_{IL} mA	Max I_{IH} mA	Max I_{OL} mA	Max I_{OH} mA	3-State Max I_{OZL} mA	3-State Max I_{OZH} mA
AD15–AD0	0.8	2.0	0.5	2.0	−0.20	0.02	24	−12.0	−0.20	0.02
A19–A16, $\overline{\mathrm{BHE}}$/S7	0.8	2.0	0.55	2.0	−0.25	0.07	63.9	−15.0	−0.07	0.07
$\overline{\mathrm{RD}}$	NA	NA	0.55	2.0	NA	NA	63.9	−15.0	−0.84	0.05
$\overline{\mathrm{DEN}}$ ($\overline{\mathrm{S0}}$), DT/$\overline{\mathrm{R}}$ ($\overline{\mathrm{S1}}$), M/$\overline{\mathrm{IO}}$ ($\overline{\mathrm{S2}}$)	0.8	2.0	0.55	2.0	−1.20	0.12	18.8	−6.6	−1.30	0.12
$\overline{\mathrm{WR}}$ ($\overline{\mathrm{LOCK}}$)	NA	NA	0.55	2.0	NA	NA	63.9	−15.0	−0.94	−0.05
$\overline{\mathrm{INTA}}$ (QS1)	NA	NA	0.5	2.4	NA	NA	19.1	−6.50	NA	NA
ALE (QS0)	NA	NA	0.5	2.4	NA	NA	19.9	−6.54	NA	NA
MN/$\overline{\mathrm{MX}}$	0.8	2.0	NA	NA	−1.6	0.04	NA	NA	NA	NA
NMI	0.77	2.0	NA	NA	−0.4	0.05	NA	NA	NA	NA
CLK, READY*	0.8	2.0	NA	NA	−3.2	0.04	NA	NA	NA	NA
INTR	0.77	2.0	NA	NA	−0.4	0.05	NA	NA	NA	NA
$\overline{\mathrm{TEST}}$	0.8	2.0	NA	NA	−0.60	0.04	NA	NA	NA	NA
RESET	0.8	2.0	NA	NA	−2.2	0.07	NA	NA	NA	NA
HOLD ($\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$), HOLDA ($\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$)	0.72	2.0	0.80	2.0	−1.60	−0.11	7.60	−7.06	NA	NA

***NOTES:**

$I_{IL} = -0.8$ mA and $I_{IH} = 0.1$ mA if a 74S244 is used at U30 for CLOCK and READY inputs.

Negative currents (−) are defined as currents flowing out of a terminal, and positive currents are defined as currents flowing into a terminal. "NA" means "not applicable."

The 8086 and 8088 chip specifications indicate that the chips have an output drive capacity of $I_{OH} = -400\ \mu$A and $I_{OL} =$ 2.5 mA (2.0 mA for the 8088); the chips' input and 3-state loading specification is $\pm 10\ \mu$A. As can be seen from the table, the 8086/8088 probe has a greater output drive capacity and presents greater input loading than the 8086 or 8088 chip.

The 8086/8088 probe does not draw any current from the user V_{CC}.

Capacitive Loading—8086/8088 Probe

- The 8086/8088 probe presents the user system with a maximum load of 70 pF (135 pF for INTR, NMI).
- All 8086/8088 probe outputs are capable of driving 0 pF while meeting all the probe's timing specifications. The 8086/8088 probe will drive larger capacitive loads, but with possible performance degradation. Derate the timing specifications by 0.04 ns/pF corresponding to input capacitance of the user system.

Coprocessor Operation—8086/8088 Probe

- During emulation with external coprocessors, a two-clock delay precedes each $\overline{RQ}$, $\overline{GT}$, and RLS pulse in MAX mode and each HOLD and HOLDA assertion in MIN mode.
- The user can choose to have the coprocessor run only during emulation or all the time. If the coprocessor runs all the time, then during interrogation mode,the coprocessor may have as much as a one-microsecond delay in addition to the two-clock delay mentioned above.
- The I²ICE system ignores a coprocessor when the probe is in the reset state. If a coprocessor asserts $\overline{RQ}$ during this time, the $\overline{RQ}/\overline{GT}$ sequence may get out of synchronization. The probe is reset when the I²ICE host software loads I²ICE probe software.

A.C. CHARACTERISTICS FOR THE I²ICE™ SYSTEM 8086 PROBE

Tables 4 through 7 provide timing information for the 8086 probe. Figures 5 through 12 define the timing symbols.

Table 4. Minimum Complexity System Timing Requirements

Min Mode Symbol	Parameter	5 MHz (8086)		10 MHz (8086-1)		8 MHz (8086-2)	
		Min ns	Max ns	Min ns	Max ns	Min ns	Max ns
TCLCL	CLK Cycle Period	200	500	100	500	125	500
TCLCH	CLK Cycle Low Time	118		53		68	
TCHCL	CLK High Time	69		39		44	
TCH1CH2	CLK Rise Time		10		10		10
TCL2CL1	CLK Fall Time		10		10		10
TDVCL(1)	Data in Setup Time	21.1		21.1(5)		21.1(20)	
TCLDX(2)	Data in Hold Time	13.5(10)		13.5(10)		13.5(10)	
TR1VCL(3, 4)	RDY Hold Time into 8284A	35		35		35	
TCLR1X(3, 4)	RDY Hold Time into 8284A	0		0		0	
TRYHCH(5)	READY Setup Time into 8086	44.5		44.5		44.5	
TCHRYX(6)	READY Hold Time into 8086	20.5		20.5(20)		20.5(20)	
TRYLCL(5)	READY Inactive to CLK	−18.5		−18.5		−18.5	
THVCH(1)	HOLD Setup Time	12.7		12.7		12.7	
TINVCH NMI(1) INTR(1) $\overline{TEST}$(1)	INTR, NMI, $\overline{TEST}$ Setup Time	50.5 + TCLCH(30) 20 21.5		50.5 + TCLCH(15) 20(15) 21.5(150		50.5 + TCLCH(15) 20(15) 21.5(15)	
TILIH	Input Rise Time (Except CLK)		20		20		20
TIHIL	Input Fall Time (Except CLK)		12		12		12

Numbers followed by parentheses deviate from the 8086 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
3. The signal at 8284 is for reference only.
4. The setup requirement, for asynchronous signal is only to guarantee recognition at the next CLK.
5. If BTHRDY = TRUE, READY must be set up 0.3 ns before the rising edge of T2.
6. If BTHRDY = TRUE, READY must be held 16.5 ns after the rising edge of T2.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 8086 PROBE (Continued)

Table 5. Minimum Complexity System Timing Responses

Min Mode Symbol	Parameter	5 MHz (8086)		10 MHz (8086-1)		8 MHz (8086-2)	
		Min ns	Max ns	Min ns	Max ns	Min ns	Max ns
TCLAV[1]	Address Valid Delay	17.5	64.5	17.5	64.5(50)	17.5	64.5(60)
TCLAX[2]	Address Hold Time	17.5		17.5		17.5	
TCLAZ[1]	Address Float Delay	14.6	61.5	14.6	61.5(40)	14.6	61.5(50)
TLHLL	ALE Width	TCLCH−17.5 (TCLCH−20)		TCLCH−17.5		TCLCH−17.5	
TCLLH[1]	ALE Active Delay		42		42(40)		42
TCHLL[1]	ALE Inactive Delay		35		35		35
TLLAX	Address Hold Time to ALE Inactive	TCHCL−8.5		TCHCL−8.5		TCHCL−8.5	
TCLDV[1]	Data Valid Delay	17.5	69.5	17.5	69.5(50)	17.5	69.5(60)
TCHDX[2]	Data Hold Time	17.5		17.5		17.5	
TWHDX	Data Hold Time after $\overline{WR}$	TCLCH−34 (TCLCH−30)		TCLCH−34 (TCLCH−25)		TCLCH−34 (TCLCH−30)	

TCVCTV
$\overline{DEN}$(READ, INTA)[1]
$\overline{DEN}$($\overline{WR}$)[1]
 	
$\overline{WR}$[1]
INTA[1]
	Control Active Delay[1]	
		15.6
		TCHCL+15.6
		
		16.9
		15.9
			
			63.5
			TCHCL+63.5
			(110)
			59.5
			55
				
				15.6
				TCHCL+15.6
				
				16.9
				15.9
					
					63.5(50)
					TCHCL+63.5
					(50)
					59.5(50)
					55(50)
						
						15.6
						TCHCL+15.6
						
						16.9
						15.9
							
							63.5
							TCHCL+63.5
							(70)
							59.5
							55

TCHCTV
M/$\overline{IO}$[1,3]
DT/$\overline{R}$[1,4]
	Control Active Delay 2	
		19
		18.4
			
			77
			73.5
				
				19
				18.4
					
					77(45)
					73.5(45)
						
						19
						18.4
							
							77(60)
							73.5(60)

TCVCTX
$\overline{DEN}$[1]
$\overline{WR}$[1]
INTA[1]
	Control Inactive Delay	
		15.6
		16.9
		15.9
			
			63.5
			59.5
			55
				
				15.6
				16.9
				15.9
					
					63.5(50)
					59.5(50)
					55(50)
						
						15.6
						16.9
						15.9
							
							63.5
							59.5
							55

TAZRL	Address Float to READ Active	−37.2(0)		−37.2(0)		−37.2(0)	
TCLRL[1]	$\overline{RD}$ Active Delay	15.9	80.5	15.9	80.5(70)	15.9	80.5
TCLRH[1]	$\overline{RD}$ Inactive Delay	15.9	70.5	15.9	70.5(60)	15.9	70.5
TRHAV	$\overline{RD}$ Inactive to Next Address Active	(Note 5)		(Note 5)		(Note 5)	
TCLHAV[1]	HLDA Valid Delay	11.3	57	11.3	57	11.3	57
TRLRH	$\overline{RD}$ Width	2TCLCL−52.5		2TCLCL−52.5 (2TCLCL−40)		2TCLCL−52.5 (2TCLCL−50)	
TWLWH	$\overline{WR}$ Width	2TCLCL−27.5		2TCLCL−27.5		2TCLCL−27.5	
TAVAL	Address Valid to ALE Low	TCLCH−47.2		TCLCH−47.2 (TCLCH−35)		TCLCH−47.2 (TCLCH−40)	
TOLOH	Output Rise Time		20		20		20
TOHOL	Output Fall Time		12		12		12

Numbers followed by parenthese deviate from the 8086 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
3. When performing consecutive I/O cycles (i.e., word I/O to an odd address), the M/IO line goes high for a short time during T4. The 8086 microprocessor keeps M/IO low between consecutive I/O cycles.
4. When performing consecutive reads to program memory, the DT/$\overline{R}$ line of the probe microprocessor (at the end of the user cable) goes high for a short time between reads. The 8086 microprocessor keeps DR/$\overline{R}$ low between consecutive reads.
5. The address data lines are only floated during T4 when $\overline{RD}$ is active.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 8086 PROBE (Continued)

Table 6. Maximum Complexity System Timing Requirements

Min Mode Symbol	Parameter	5 MHz (8086)		10 MHz (8086-1)		8 MHz (8086-2)	
		Min	Max ns	Min ns	Max ns	Min ns	Max ns
TCLCL	CLK Cycle Period	200	500	100	500	125	500
TCLCH	CLK Low Time	118		60(53)		68	
TCHCL	CLK High Time	69		39		44	
TCH1CH2	CLK Rise Time		10		10		10
TCL2CL1	CLK Fall Time		10		10		10
TDVCL(1)	Data in Setup Time	21.1		21.1(5)		21.1(20)	
TCLDX(2)	Data in Hold Time	13.5(10)		13.5(10)		13.5(10)	
TR1VCL(3, 4)	RDY Setup Time into 8284A	35		35		35	
TCLR1X(3, 4)	RDY Hold Time into 8284A	0		0		0	
TRYHCH(5)	READY Setup Time into 8086	44.5		44.5		44.5	
TCHRYX(6)	READY Hold Time into 8086	20.5		20.5(20)		20.5(20)	
TRYLCL(5)	READY Inactive to CLK	−18.5		−18.5		−18.5	
TINVCH NMI(1) INTR(1) $\overline{\text{TEST}}$(1)	Setup Time for Recognition (INTR, NMI, $\overline{\text{TEST}}$)	50.5 + TCLCH(30) 20 21.5		50.5 + TCLCH(15) 20(15) 21.5(15)		50.5 + TCLCH(15) 20(15) 21.5(15)	
TGVCH(1)	$\overline{\text{RQ}}/\overline{\text{GT}}$ Setup Time	12.7		12.7(12)		12.7	
TCHGX(2)	$\overline{\text{RQ}}$ Hold Time into 8086	16.1	16.1			16.1	
TILIH	Input Rise Time (Except CLK)		20		20		20
TIHIL	Input Fall Time (Except CLK)		12		12		12

Numbers followed by parentheses deviate from the 8086 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
3. The signal at 8284 or 8288 is for reference only.
4. The setup requirement, for asynchronous signal is only to guarantee recognition at the next CLK.
5. If BTHRDY = TRUE, READY must be set up 0.3 ns before the rising edge of T2.
6. If BTHRDY = TRUE, READY must be held 16.5 ns after the rising edge of T2.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 8086 PROBE (Continued)

Table 7. Maximum Complexity System Timing Responses

Min Mode Symbol	Parameter	5 MHz (8086)		10 MHz (8086-1)		8 MHz (8086-2)	
		Min ns	Max ns	Min ns	Max ns	Min ns	Max ns
TCLML(1)	Command Active Delay	10	35	10	35	10	35
TCLMH(1)	Command Inactive Delay	10	35	10	35	10	35
TRYHSH(2, 3, 4)	READY Active to Status Passive		37.5		37.5		37.5
TCHSV(4)	Status Active Delay	17	66.5	17	66.5(45)	17	66.5(60)
TCLSH(4)	Status Inactive Delay	10.5	42.5	10.5	42.5	10.5	42.5
TCLAV(4)	Address Valid Delay	17.5	64.5	17.5	64.5(50)	17.5	64.5(60)
TCLAX(5)	Address Hold Time	17.5(10)		17.5(10)		17.5(10)	
TCLAZ(4)	Address Float Delay	14.6	61.5	14.6	61.5(40)	14.6	61.5(50)
TSVLH(1)	Status Valid to ALE High		15		15		15
TSVMCH(1)	Status Valid to MCE High		15		15		15
TCLLH(1)	CLK Low to ALE Valid		15		15		15
TCLMCH(1)	CLK Low to MCE High		15		15		15
TCHLL(1)	ALE Inactive Delay		15		15		15
TCLMCL(1)	MCE Inactive Delay		15		15		15
TCLDV(4)	Data Valid Delay	17.5	69.5	17.5	69.5(50)	17.5	69.5(60)
TCHDX(5)	Data Hold Time	17.5		17.5		17.5	
TCVNV(1)	Control Active Delay	5	45	5	45	5	45
TCVNX(1)	Control Inactive Delay	10	45	10	45	10	45
TAZRL	Address Float to Read Active	−37.2(0)		−37.2(0)		−37.2(0)	
TCLRL(4)	$\overline{RD}$ Active Delay	15.9	80.5	15.9	80.5(70)	15.9	80.5
TCLRH(4)	$\overline{RD}$ Inactive Delay	15.9	70.5	15.9	70.5(60)	15.9	70.5
TRHAV	$\overline{RD}$ Inactive to Next Address Active	(Note 6)		(Note 6)		(Note 6)	
TCHDTL(1)	Direction Control Active Delay		50		50		50
TCHDTH(1)	Direction Control Inactive Delay		30		30		30
TCLGL(4)	$\overline{GT}$ Active Delay	12.9	54.5	12.9	54.5(45)	12.9	54.5(50)
TCLGH(4)	$\overline{GT}$ Inactive Delay	14.9	65	14.9	65(45)	14.9	65(50)
TRLRH	$\overline{RD}$ Width	2TCLCL−52.5		2TCLCL−52.5 (2TCLCL−40)		2TCLCL−52.5 (2TCLCL−50)	
TOLOH	Output Rise Time		20		20		20
TOHOL	Output Fall Time		12		12		12

Numbers followed by parentheses deviate from the 8088 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. The signal at 8284 or 8288 is for reference only.
2. If BTHRDY = TRUE, READY must be set up 0.3 ns before the rising edge of T2.
3. For BTHRDY = TRUE, TRYHSH = TRYHCH + 47.
4. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
5. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
6. The address data lines are only floated during T4 when RD is active.

A.C. CHARACTERISTICS FOR THE I²ICE™ SYSTEM 8088 PROBE

Tables 8 through 11 provide timing information for the 8088 probe. Figures 5 through 12 define the timing symbols.

Table 8. Minimum Complexity System Timing Requirements

Min Mode Symbol	Parameter	5 MHz (8088)		8 MHz (8088-1)	
		Min ns	Max ns	Min ns	Max ns
TCLCL	CLK Cycle Period	200	500	125	500
TCLCH	CLK Low Time	118		68	
TCHCL	CLK High Time	69		44	
TCH1CH2	CLK Rise Time		10		10
TCL2CL1	CLK Fall Time		10		10
TDVCL(1)	Data in Setup Time	21.1		21.1(20)	
TCLDX(2)	Data in Hold Time	13.5(10)		13.5(10)	
TR1VCL(3, 4)	RDY Setup Time into 8284	35		35	
TCLR1X(3, 4)	RDY Hold Time into 8284	0		0	
TRYHCH(5)	READY Setup Time into 8088	57.8		57.8	
TCHRYX(6)	READY Hold Time into 8088	20.5		20.5(20)	
TRYLCL(5)	READY Inactive to CLK	−16.5		−16.5	
THVCH(1)	Hold Setup Time	12.7		12.7	
TINVCH NMI(1)	INTR, NMI, $\overline{\text{TEST}}$ Setup Time	50.5 + TCLCH(30)		50.5 + TCLCH(15)	
INTR(1)		26		26(15)	
$\overline{\text{TEST}}$(1)		27.5		27.5(15)	
TILIH	Input Rise Time		20		20
TIHIL	Input Fall Time		12		12

Numbers followed by parentheses deviate from the 8088 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
3. The signal at 8284 is for reference only.
4. The setup requirement, for asynchronous signal is only to guarantee recognition at the next CLK.
5. For BTHRDY = TRUE, READY must be set up 0.3 ns prior to the rising edge of T2.
6. For BTHRDY = TRUE, READY must be held 16.5 ns after the rising edge of T2.

A.C. CHARACTERISTICS FOR THE I²ICE™ SYSTEM 8088 PROBE (Continued)

Table 9. Minimum Complexity System Timing Responses

Min Mode Symbol	Parameter	5 MHz (8088)		8 MHz (8088-2)	
		Min ns	Max ns	Min ns	Max ns
TCLAV(1)	Address Valid Delay	17.5	72	17.5	72(60)
TCLAX(2)	Address Hold Time	17.5		17.5	
TCLAZ(1)	Address Float Delay	13.6	61.5	13.6	61.5(50)
TLHLL	ALE Width	TCLCH − 17.5 (TCLCH − 20)		TCLCH − 17.5	
TCLLH(1)	ALE Active Delay		41		41
TCHLL(1)	ALE Inactive Delay		35		35
TLLAX	Address Hold Time to ALE Inactive	TCHCL − 8.5		TCLCH − 8.5	
TCLDV(1)	Data Valid Delay	17.5	70.5	17.5	70.5(60)
TCHDX(2)	Data Hold Time	17.5		17.5	
TWHDX	Data Hold Time after $\overline{\text{WR}}$	TCLCH − 34 (TCLCH − 30)		TCLCH − 34 (TCLCH − 30)	
TCVCTV	Control Active Delay 1				
$\overline{\text{DEN}}$(RD, INTA)(1)		15.6	63.5	15.6	63.5
$\overline{\text{DEN}}$(WRITE)(1)		TCHCL + 13.6	TCHCL + 63.5 (110)	TCHCL + 13.6	TCHCL + 63.5 (70)
$\overline{\text{WR}}$(1)		16.9	59.5	16.9	59.5
INTA(1)		15.9	55	15.9	55
TCHCTV	Control Inactive Delay 2				
SS0(1)		16.3	104	16.3	104(60)
IO/$\overline{\text{M}}$(1, 3)		19.1	81	19.1	81(60)
DT/$\overline{\text{R}}$(1, 4)		18.3	77.5	18.3	77.5(60)
TCVCTX	Control Inactive Delay				
$\overline{\text{DEN}}$(1)		15.6	63.5	15.6	63.5
$\overline{\text{WR}}$(1)		16.9	59.5	16.9	59.5
INTA(1)		15.9	55	15.9	55
TAZRL	Address Float to READ Active	−37.2(0)		−37.2(0)	
TCLRL(1)	$\overline{\text{RD}}$ Active Delay	15.9	110.5	15.9	110.5(100)
TCLRH(1)	$\overline{\text{RD}}$ Inactive Delay	15.9	90.5	15.9	90.5(80)
TRHAV	$\overline{\text{RD}}$ Inactive to Next Address Active	(Note 5)		(Note 5)	
TCLHAV(1)	HLDA Valid Delay	13.3	57	13.3	57
TRLRH	$\overline{\text{RD}}$ Width	2TCLCL − 82.5 (2TCLCL − 75)		2TCLCL − 82.5 (2TCLCL − 50)	
TWLWH	$\overline{\text{WR}}$ Width	2TCLCL − 27.5		2TCLCL − 27.5	
TAVAL	Address Valid to ALE Low	TCLCH − 52.2		TCLCH − 52.2 (TCLCH − 40)	
TOLOH	Output Rise Time		20		20
TOHOL	Output Fall Time		12		12

Numbers followed by parentheses deviate from the 8088 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
3. When performing consecutive I/O cycles (i.e., word I/O to an odd address), the M/IO line goes high for a short time during T4. The 8088 microprocessor keeps IO/M low between consecutive I/O cycles.
4. When performing consecutive reads to program memory, the DT/$\overline{\text{R}}$ line of the probe microprocessor (at the end of the user cable) goes high for a short time between reads. The 8088 microprocessor keeps DR/$\overline{\text{R}}$ low between consecutive reads.
5. The address data lines are only floated during T4 when $\overline{\text{RD}}$ is active.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 8088 PROBE (Continued)

Table 10. Maximum Complexity System Timing Requirements

Min Mode Symbol	Parameter	5 MHz (8088)		8 MHz (8088-2)	
		Min ns	Max ns	Min ns	Max ns
TCLCL	CLK Cycle Period	200	500	125	500
TCLCH	CLK Low Time	118		68	
TCHCL	CLK High Time	69		44	
TCH1CH2	CLK Rise Time		10		10
TCL2CL1	CLK Fall Time		10		10
TDVCL(1)	Data in Setup Time	21.1		21.1(20)	
TCLDX(2)	Data in Hold Time	13.5(10)		13.5(10)	
TR1VCL(3, 4)	RDY Setup Time into 8284	35		35	
TCLR1X(3, 4)	RDY Hold Time into 8284	0		0	
TRYHCH(5)	READY Setup Time into 8088	57.8		57.8	
TCHRYX(6)	READY Hold Time into 8088	20.5		20.5(20)	
TRYLCL(5)	READY Inactive to CLK	−16.5		−16.5	
TINVCH NMI(1)	Setup Time for Recognition (INTR, NMI, $\overline{TEST}$)	50.5 + TCLCH(30)		50.5 + TCLCH(15)	
INTR(1)		26		26(15)	
$\overline{TEST}$(1)		27.5		27.5(15)	
TGVCH(1)	$\overline{RQ}/\overline{GT}$ Setup Time	12.7		12.7	
TCHGX(2)	$\overline{RQ}$ Hold Time into 8088	16.1		16.1	
TILIH	Input Rise Time (Except CLK)		20		20
TIHIL	Input Fall Time (Except CLK)		12		12

Numbers followed by parentheses deviate from the 8088 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
2. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
3. The signal at 8284 or 8288 is for reference only.
4. The setup requirement, for asynchronous signal is only to guarantee recognition at the next CLK.
5. If BTHRDY = TRUE, READY must be set up 0.3 ns before the rising edge of T2.
6. If BTHRDY = TRUE, READY must be held 16.5 ns after the rising edge of T2.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 8088 PROBE (Continued)

Table 11. Maximum Complexity System Timing Responses

Min Mode Symbol	Parameter	5 MHz (8088)		8 MHz (8088-2)	
		Min ns	Max ns	Min ns	Max ns
TCLML(1)	Command Active Delay	10	35	10	35
TCLMH(1)	Command Inactive Delay	10	35	10	35
TRYHSH(2, 3, 4)	READY Active to Status Passive		37.5		37.5
TCHSV(4)	Status Active Delay	16.3	70.5	16.3	70.5 (60)
TCLSH(4)	Status Inactive Delay	10.5	42.5	10.5	42.5
TCLAV(4)	Address Valid Delay	17.5	72	17.5	72 (60)
TCLAX(5)	Address Hold Time	17.5		17.5	
TCLAZ(4)	Address Float Delay	13.6	61.5	13.6	61.5 (50)
TSVLH(1)	Status Valid to ALE High		15		15
TSVMCH(1)	Status Valid to MCE High		15		15
TCLLH(1)	CLK Low to ALE Valid		15		15
TCLMCH(1)	CLK Low to MCE High		15		15
TCHLL(1)	ALE inactive Delay		15		15
TCLMCL(1)	MCE Inactive Delay		15		15
TCLDV(4)	Data Valid Delay	17.5	70.5	17.5	70.5 (60)
TCHDX(5)	Data Hold Time	17.5		17.5	
TCVNV(1)	Control Active Delay	5	45	5	45
TCVNX(1)	Control Inactive Delay	10	45	10	45
TAZRL	Address Float to READ Active	−37.2(0)		−37.2(0)	
TCLRL(4)	$\overline{RD}$ Active Delay	15.9	110.5	15.9	110.5 (100)
TCLRH(4)	$\overline{RD}$ Inactive Delay	15.9	90.5	15.9	90.5 (80)
TRHAV	$\overline{RD}$ Inactive to Next Address	(Note 6)		(Note 6)	
TCHDTL(1)	Direction Control Active Delay		50		50
TCHDTH(1)	Direction Control Inactive Delay		30		30
TCLGL(4)	$\overline{GT}$ Active Delay	12.9	54.5	12.9	54.5 (50)
TCLGH(4)	$\overline{GT}$ Inactive Delay	14.9	65	14.9	65 (50)
TRLRH	$\overline{RD}$ Width	2TCLCL−82.5 (2TCLCL−75)		2TCLCL−82.5 (2TCLCL−50)	
TOLOH	Output Rise Time		20		20
TOHOL	Output Fall Time		12		12

Numbers followed by parentheses deviate from the 8088 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTES:
1. The signal at 8284 or 8288 is for reference only.
2. If BTHRDY = TRUE, READY must be set up 0.3 ns before the rising edge of T2.
3. For BTHRDY = TRUE, TRYHSH = TRYHCH + 47.
4. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 2.5 ns to the timings.
5. Timings are calculated with a 74F244 as the buffer for CLOCK or READY. If a 74S244 is used, add 0.7 ns to the timings.
6. The address data lines are only floated during T4 when RD is active.

8086/8088 PROBE WAVEFORMS

MINIMUM MODE

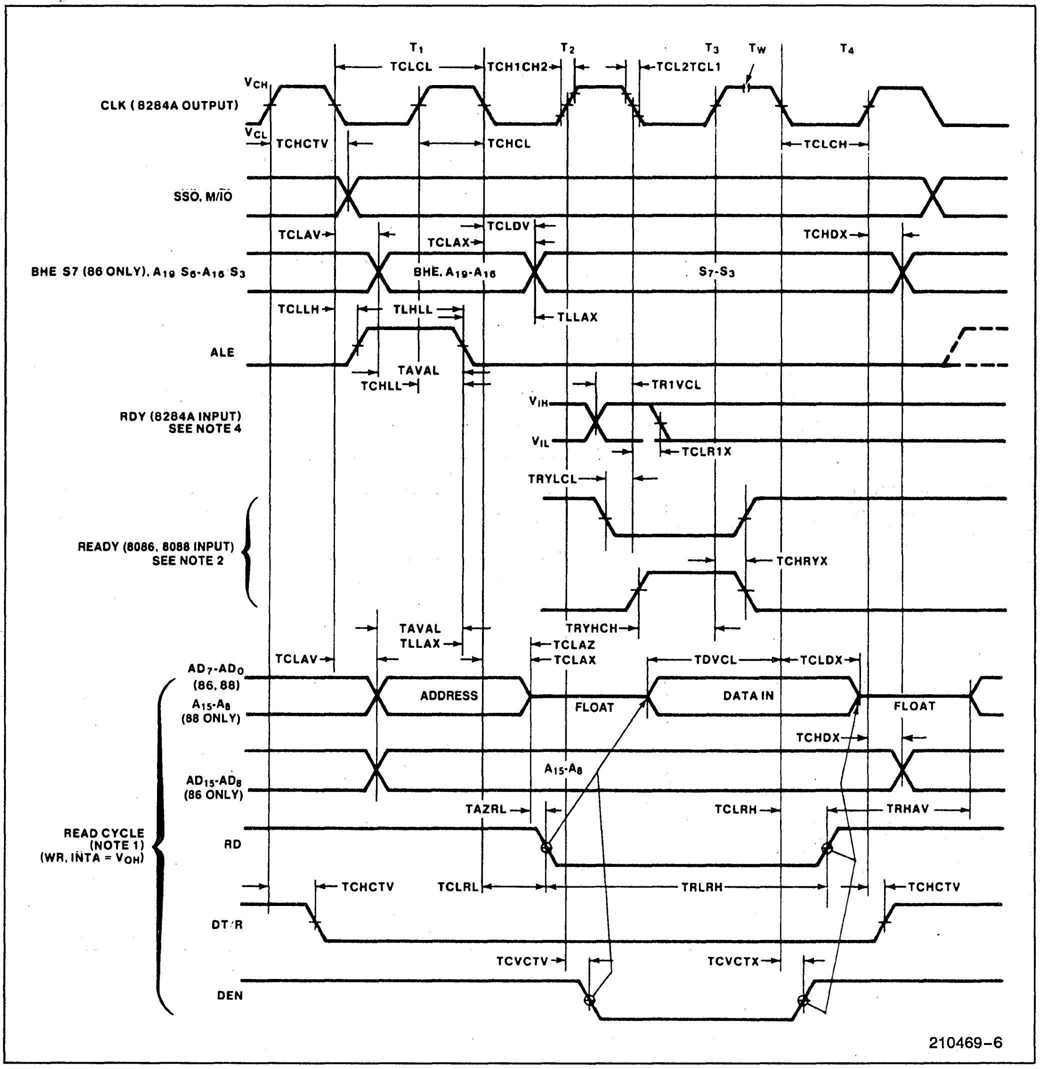
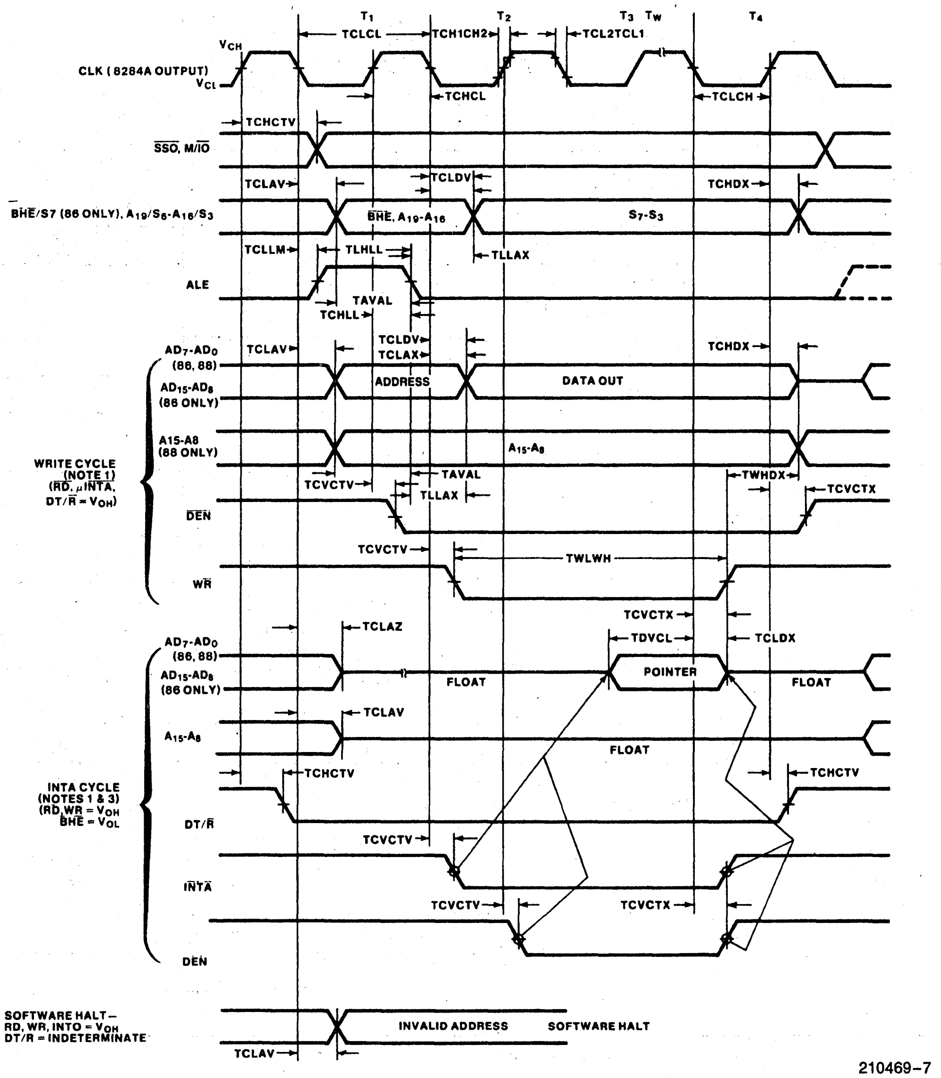


Figure 5. (Continued on next page)

8086/8088 PROBE WAVEFORMS (Continued)

MINIMUM MODE (Continued)



NOTES:
1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. READY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted. Timing shown is for BTHRDY = FALSE; see Figure 12 for BTHRDY = TRUE.
3. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control signals shown for second INTA cycle.
4. Signals at 8284A aer shown for reference only.

Figure 5. (Continued)

8086/8088 PROBE WAVEFORMS (Continued)

MAXIMUM MODE

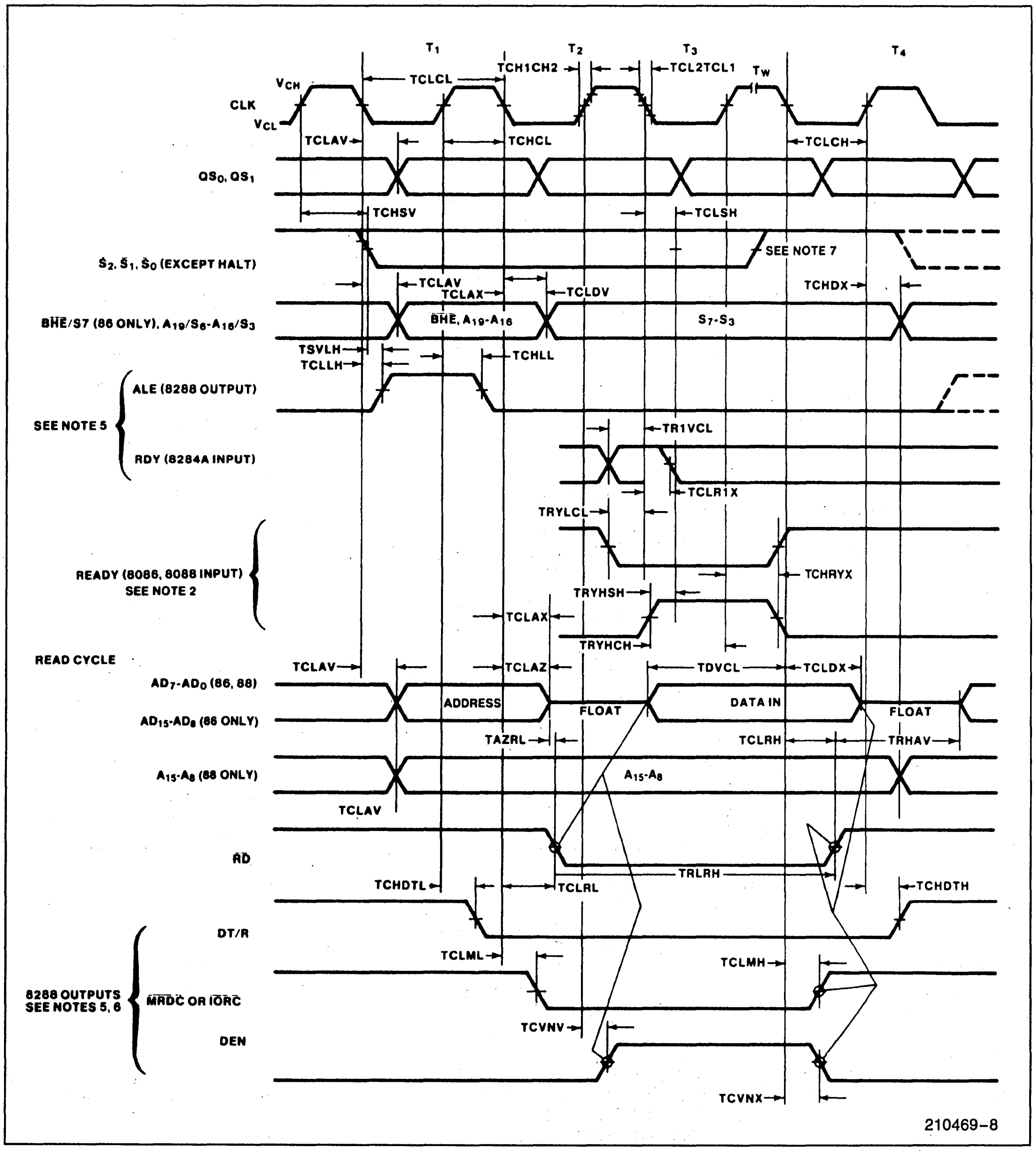
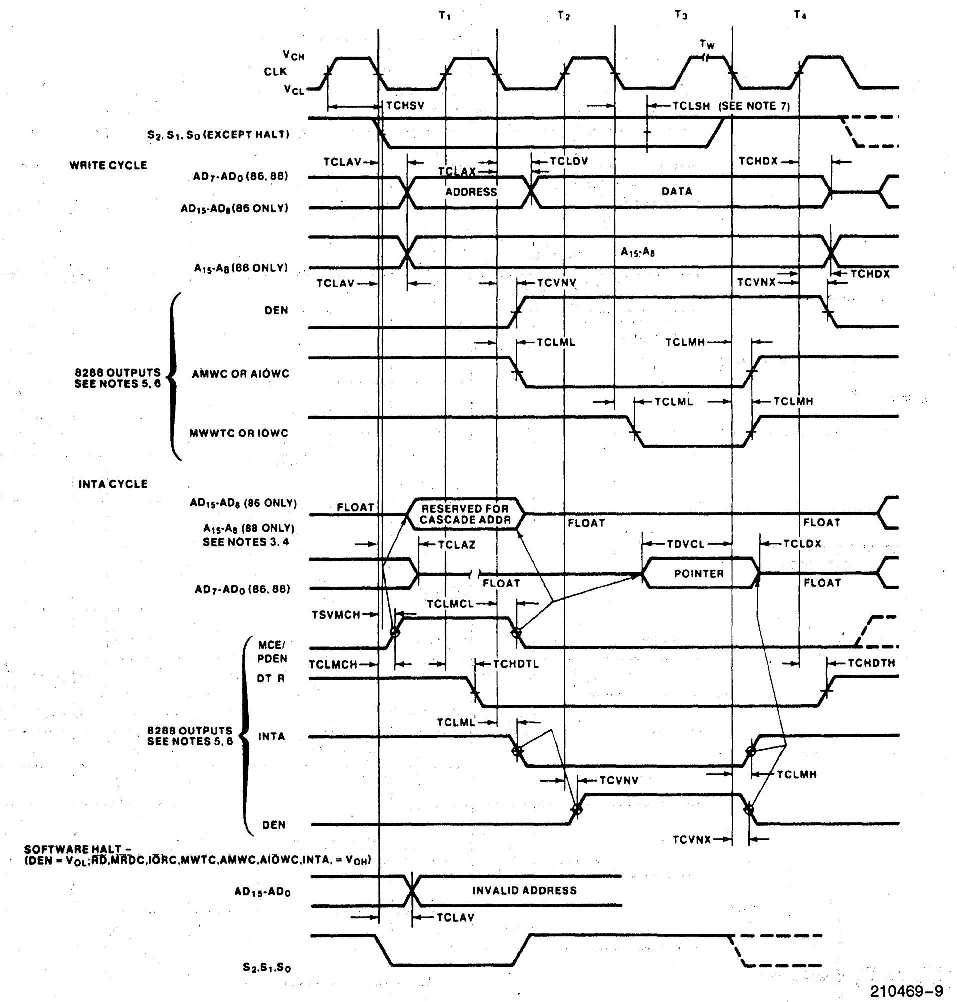


Figure 6. (Continued on next page)

8086/8088 PROBE WAVEFORMS (Continued)

MAXIMUM MODE (Continued)

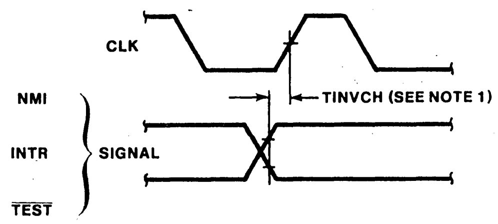


NOTES:
1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. READY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted. Timing shown is for BTHRDY = FALSE; see Figure 12 for BTHRDY = TRUE.
3. Cascade address is valid between first and second INTA cycle.
4. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control for pointer address is shown for second INTA cycle.
5. Signals at 8284A and 8288 are shown for reference only.
6. The issuance of the 8288 command and control signals ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOC}$, $\overline{INTA}$, and $\overline{DEN}$) lags the active high 8288 CEN.
7. Status inactive in state just prior to T_4.

Figure 6. (Continued)

8086/8088 PROBE WAVEFORMS (Continued)

ASYNCHRONOUS SIGNAL RECOGNITION

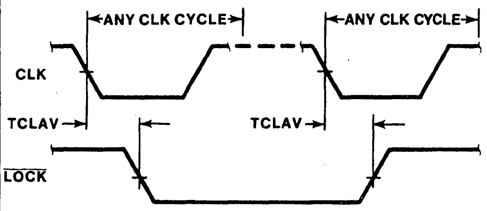


210469-10

NOTE:
1. Setup requirements for asynchronous signals only to guarantee recognition at next CLK.

Figure 7

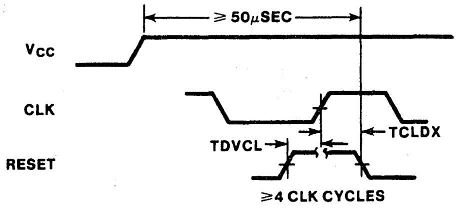
BUS LOCK SIGNAL TIMING (MAXIMUM MODE ONLY)



210469-11

Figure 8

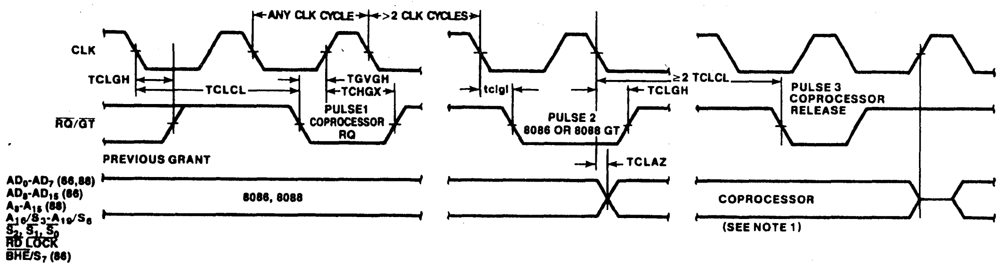
RESET TIMING



210469-12

Figure 9

REQUEST/GRANT SEQUENCE TIMING (MAXIMUM MODE ONLY)



210469-13

NOTE:
1. The coprocessor may not drive the buses outside the region shown without risking contention.

Figure 10

8086/8088 PROBE WAVEFORMS (Continued)

HOLD/HOLD ACKNOWLEDGE TIMING (MINIMUM MODE ONLY)

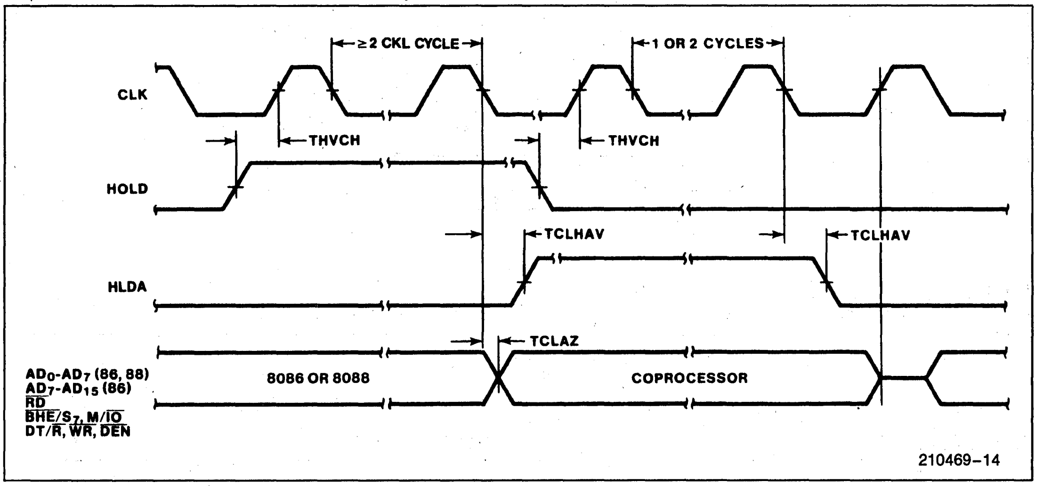


Figure 11

READY TIMING FOR BTHRDY = TRUE

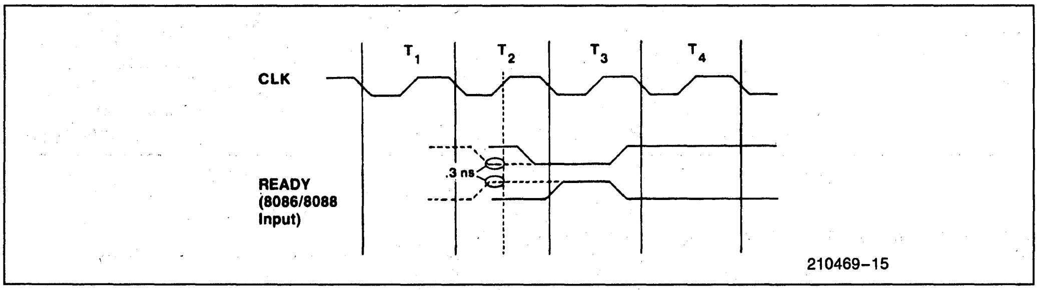


Figure 12

I2ICE™ SYSTEM 80186/80188 PROBE HIGHLIGHTS

- One megabyte addressing.
- Supports standard and queue status modes.
- Keyword access to the internal peripheral control block and the relocation register.

Table 12. I2ICE™ 80186/80188 User Interface—D.C. Characteristics

Pin Name	Input Voltage		Output Voltage		Input Current		Output Current	
	Max V_{IL} V	Min V_{IH} V	Max V_{OL} V	Min V_{OH} V	Max I_{IL} mA	Max I_{IH} mA	Max I_{OL} mA	Max I_{OH} mA
$\overline{X1}$	0.6	3.0			−0.0055	0.0055		
$\overline{RES}$	0.6	3.0			−0.0114	0.0082		
$\overline{TEST}$	0.6	2.2			−0.6	0.07		
TMRIN0, TMRIN1, DRQ0 DRQ1, NMI	0.6	2.2			−0.4	0.05		
HOLD	0.6	2.2			−1.6	0.04		
INT0, INT1	0.6	2.2			−0.21	0.03		
ARDY, SRDY	0.6	2.2			−2.0	0.07		
INT2/$\overline{INTA0}$, INT3/$\overline{INTA1}$	0.6	2.2	0.6	2.2	−0.21	0.03	2.0	−0.4
AD0–AD15	0.6	2.2	0.6	2.2	−0.45	0.1	64	−15
CLKOUT, A19/S6-A/16/S3, TMROUT0, TMROUT1, $\overline{BHE}$/S7, ALE/QS0, $\overline{WR}$/QS1, $\overline{RD}$/$\overline{QSMD}$, $\overline{LOCK}$, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, HLDA, $\overline{UCS}$, $\overline{LCS}$, $\overline{MCS0-3}$, $\overline{PCS0-4}$, $\overline{PCS5}$/A1, $\overline{PCS6}$/A2, DT/$\overline{R}$, $\overline{DEN}$			0.6	2.2			64	−15
RESET			0.6	2.2			9.1	−15

NOTES:
1. Negative currents (−) are defined as currents flowing out of a terminal, and positive currents are defined as currents flowing into a terminal.
2. The 80186 and 80188 chip specifications indicate that the chips have an output drive capacity of I_{OH} = −400 μA and I_{OL} = 2.0 mA (2.5 mA for $\overline{S0}$–$\overline{S2}$); the chips' input and 3-state loading specification is ±10 μA. As can be seen from the table, the 80186/80188 probe has a greater output drive capacity than the 80186 or 80188 chip; it also presents greater input loading than the 80186 or 80188 chip (except for $\overline{X1}$ and the I_{IH} value for $\overline{RES}$).
3. The 80186/80188 probe does not draw any current from the user V_{CC}.

Capacitive Loading—80186/80188 Probe

- The 80186/80188 probe presents the user system with a maximum load of 80 pF (120 pF for ARDY, SRDY, and $\overline{\text{TEST}}$).

Coprocessor Operation—80186/80188 Probe

- The user can choose to have the coprocessor run only during emulation or all the time. If the coprocessor runs all the time, then during interrogation mode, the coprocessor may have as much as a one-microsecond delay.

AC CHARACTERISTICS FOR THE I2ICE™ SYSTEM 80186/80188 PROBE

All timings are measured at 1.5V unless otherwise noted. Tables 13 through 17 provide timing information for the 80186/80188 probe. Figures 13 through 15 define the timing symbols.

Table 13. Timing Requirements

Symbol	Parameter	8 MHz		Test Conditions
		Min ns	Max ns	
TDVCL	Data in Setup (A/D)	41.0 (20)		
TCLDX	Data in Hold (A/D)	−1.6		
TARYHCH	AREADY Active Setup Time	40.2 (20)		
TARYLCL	AREADY Inactive Setup Time	55.2 (35)		
TCHARYX	AREADY Hold Time	2.7		
TSYRCL	SREADY Transition Setup Time	45.2 (35)		
TCLSRY	SREADY Transition Hold Time	2.7		
THVCL	Hold Setup	42.5 (25)		
TINVCH	NMI Setup	76.0 (25)		NMI Only
TINVCH	INT0–INT3 Setup	37.0 (25)		INT0–INT3 Only
TINVCH	$\overline{\text{TEST}}$, TIMERIN Setup	46.0 (25)		All Others
TINVCL	DRQ0, DRQ1 Setup	41.0 (25)		

Numbers followed by parentheses deviate from the 80186/80188 chip specification; the 1984 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 80186/80188 PROBE
(Continued)

Table 14. Master Interface Timing Responses

Symbol	Parameter	8 MHz Min ns	8 MHz Max ns	Test Conditions
TCLAV	Address Valid Delay	−2.2 (5)	51.2 (44)	
TCLAX	Address Hold	−2.2 (10)		
TCLAZ	Address Float Delay	15.1	99.7 (35)	During HLDA Cycles Only
TCLAZ	Address Float Delay	14.8	52.0 (35)	During $\overline{\text{RD}}$ Cycles only
TCLAZ	Address Float Delay	−51.2 (TCLAX)	54.7 (35)	During $\overline{\text{INTA}}$ Cycles
TCHCZ	Command Lines Float Delay		151.7 (45)	
TCHCV	Command Lines Valid Delay (after Float)		66.2 (55)	
TLHLL	ALE Width	TCLCL−13.6 (TCLCL−35)		
TCHLH	ALE Active Delay		21.2	
TCHLL	ALE Inactive Delay		41.3 (35)	
TLLAX	Address Hold to ALE	TCHCL−34 (TCHCL − 25)		
TCLDV	Data Valid Delay	−2.2 (10)	43.2	
TCLDOX	Data Hold Time	−2.2 (10)		
TWHDX	Data Hold after $\overline{\text{WR}}$	TCLCL − 28.2		
TCVCTV	Control Active Delay(1)	−2.2 (5)	69.2	For $\overline{\text{DEN}}$
TCVCTV	Control Active Delay(1)	14.1	35.5	For $\overline{\text{WR}}$
TCVCTV	Control Active Delay(1)	−4.0 (5)	60.2	For $\overline{\text{INTA}}$
TCHCTV	Control Active Delay(2)	−2.2 (10)	62.2 (55)	
TCVCTX	Control Inactive Delay	−2.2 (5)	62.2 (55)	For $\overline{\text{DEN}}$
TCVCTX	Control Inactive Delay	14.1	35.5	For $\overline{\text{WR}}$
TCVCTX	Control Inactive Delay	−4.0 (10)	3.2	For $\overline{\text{INTA}}$
TCVDEX	$\overline{\text{DEN}}$ Inactive Delay (Non-Write Cycle)		69.2	
TAZRL	Address Float to $\overline{\text{RD}}$ Active	−35.6 (0)		

Numbers followed by parentheses deviate from the 80186/80186 chip specification; the 1984 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

A.C. CHARACTERISTICS FOR THE I2ICE™ SYSTEM 80186/80188 PROBE
(Continued)

Table 14. Master Interface Timing Responses (Continued)

Symbol	Parameter	8 MHz Min ns	8 MHz Max ns	Test Conditions
TCLRL	$\overline{RD}$ Active Delay	14.1	35.5	
TCLRH	$\overline{RD}$ Inactive Delay	14.1	35.5	
TRHAV	$\overline{RD}$ Inactive to Address Active	TCLCL−37.2		
TCLHAV	HLDA Valid Delay	2.8 (10)	54.7 (50)	
TRLRH	$\overline{RD}$ Width	2TCLCL−19.3		
TWLWH	$\overline{WR}$ Width	2TCLCL		
TAVAL	Address Valid to ALE Low	TCLCH−26.6 (TCLCH−25)		
TCHSV	Status Active Delay	2.8 (10)	57.2 (55)	
TCLSH	Status Inactive Delay	2.8 (10)	62.2 (55)	
TCLTMV	Timer Output Delay		59.2	
TCLRO	Reset Delay		52.5	
TCHQSV	Queue Status Delay		34.2	

Numbers followed by parentheses deviate from the 80186/80186 chip specification; the 1984 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

Table 15. Chip-Select Timing Responses

Symbol	Parameter	8 MHz Min ns	8 MHz Max ns
TCLCSV	Chip-Select Active Delay		65.2
TCXCSX	Chip-Select Hold from Command Inactive	17.8 (35)	
TCHCSX	Chip-Select Inactive Delay	−2.2 (5)	42.2 (35)

Numbers followed by parentheses deviate from the 80186/80188 chip specification; the 1984 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

A.C. CHARACTERISTICS FOR THE I²ICE™ SYSTEM 80186/80188 PROBE
(Continued)

Table 16. CLKIN Requirements

Symbol	Parameter	8 MHz Min ns	8 MHz Max ns	Test Conditions
TCKIN	CLKIN Period	62.5	250.0	
TCKHL	CLKIN Fall Time		10.0	3.5V to 1.0V
TCKLH	CLKIN Rise Time		10.0	1.0V to 3.5V
TCLCK	CLKIN Low Time	25.0		
TCHCK	CLKIN High Time	25.0		

Table 17. CLKOUT Timing

Symbol	Parameter	8 MHz Min ns	8 MHz Max ns	Test Conditions
TCICO	CLKIN to CLKOUT Skew		97 (50)	
TCLCL	CLKOUT Period	125	500	
TCLCH	CLKOUT Low Time	1/2TCLCL−7.5		
TCHCL	CLKOUT High Time	1/2TCLCL−7.5		
TCH1CH2	CLKOUT Rise Time		15	1.0V to 3.5V
TCL2CL1	CLKOUT Fall Time		15	3.5V to 1.0V

Numbers followed by parentheses deviate from the 80186/80188 chip specification; the 1984 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

80186/80188 PROBE WAVEFORMS

MAJOR CYCLE TIMING

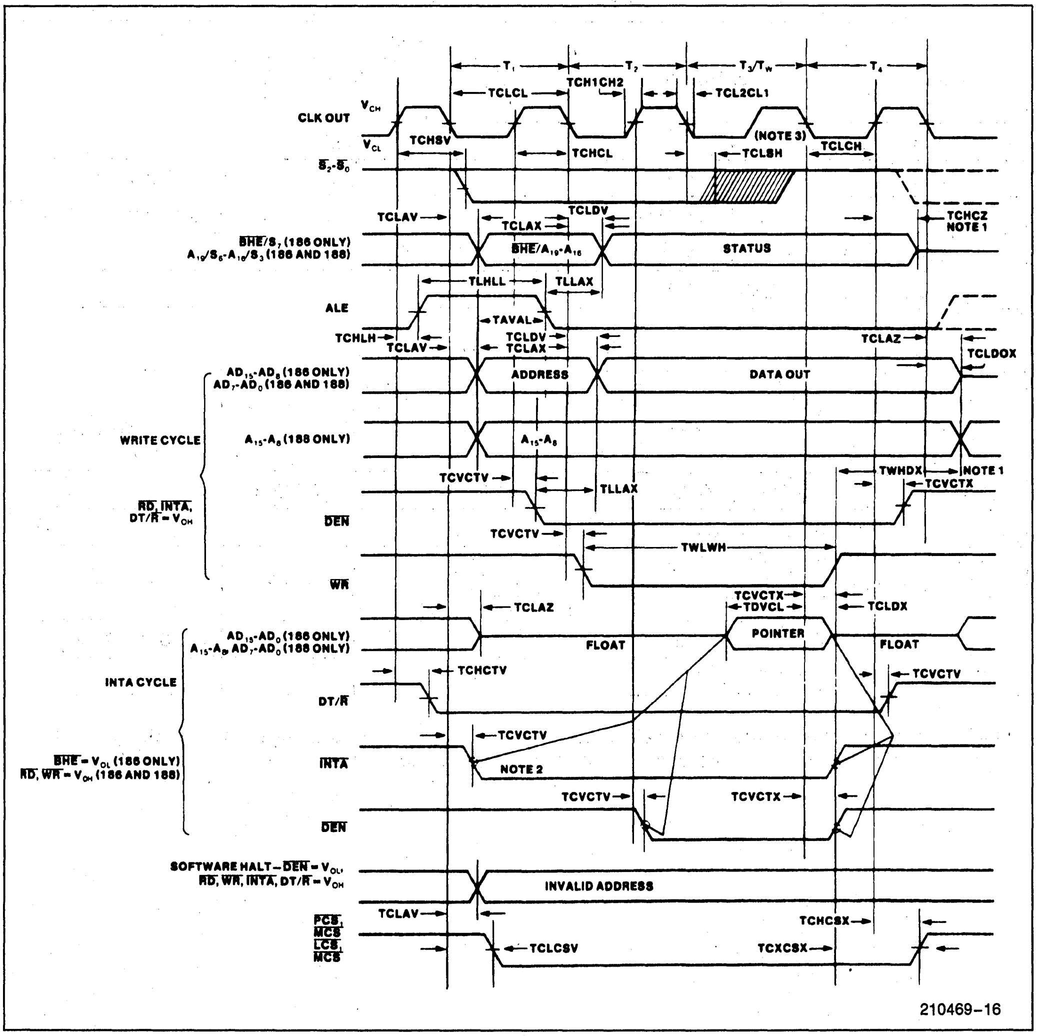
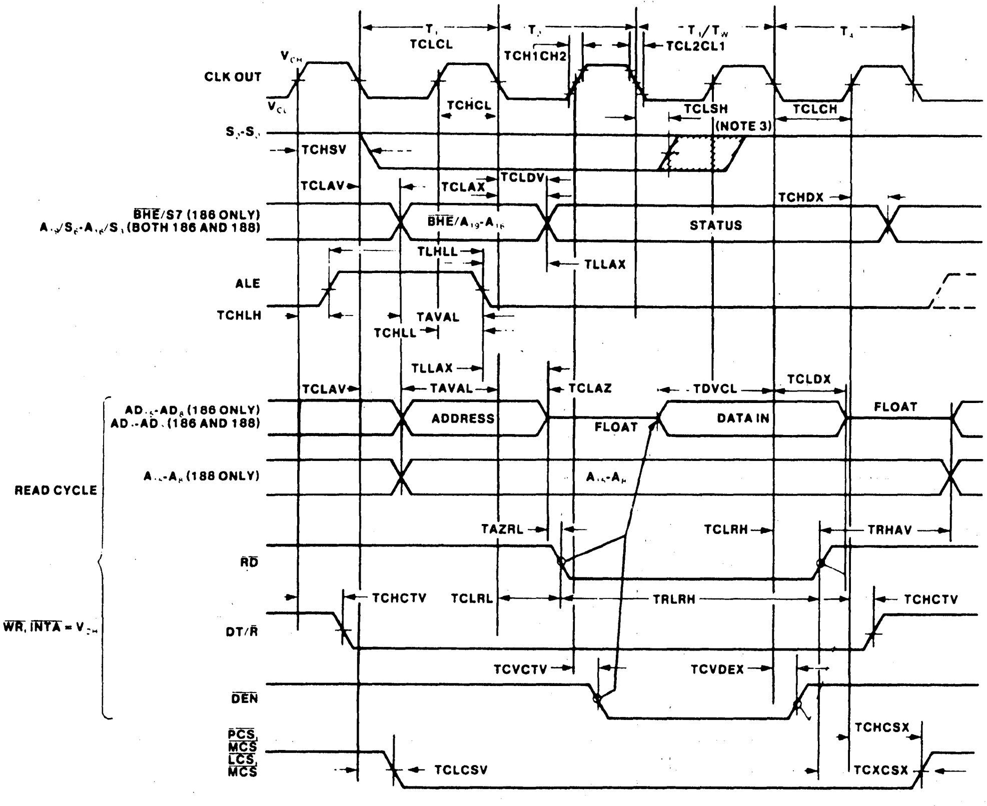


Figure 13 (Continued on next page)

80186/80188 PROBE WAVEFORMS (Continued)

MAJOR CYCLE TIMING (Continued)



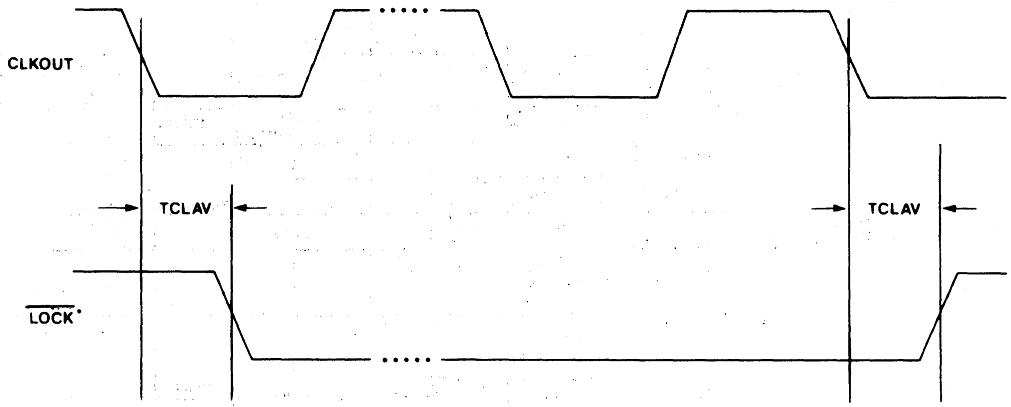
210469-17

NOTES:
1. Following a Write cycle, the Local Bus is floated by the 80186/80188 only when the 80186/80188 enters a "Hold Acknowledge" state.
2. INTA occurs one clock later in RMX-mode.
3. Status inactive just prior to T.

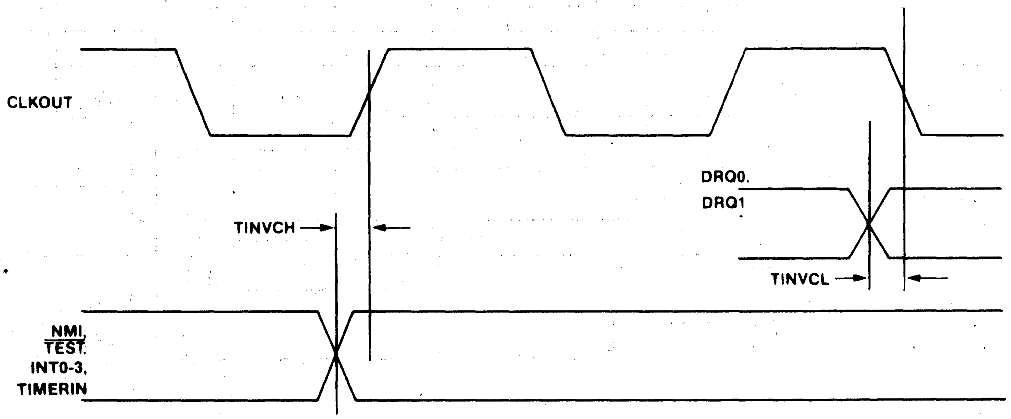
Figure 13 (Continued)

80186/80188 PROBE WAVEFORMS (Continued)

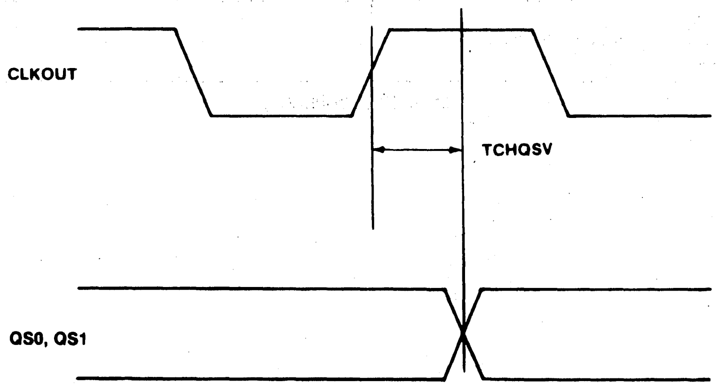
MAJOR CYCLE TIMING (Continued)



210469–18



210469–19

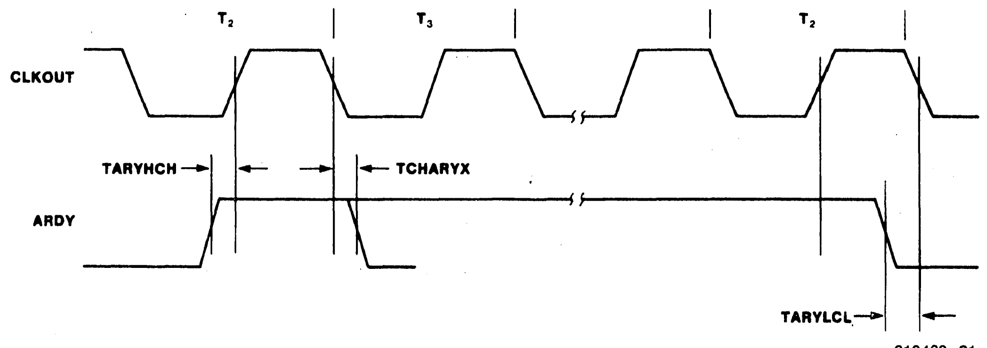


210469–20

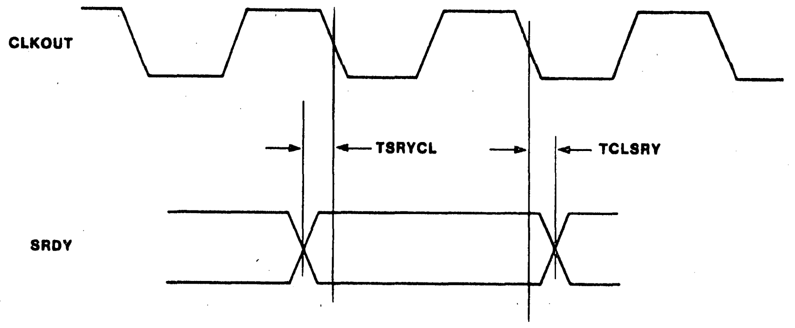
Figure 13 (Continued)

80186/80188 PROBE WAVEFORMS (Continued)

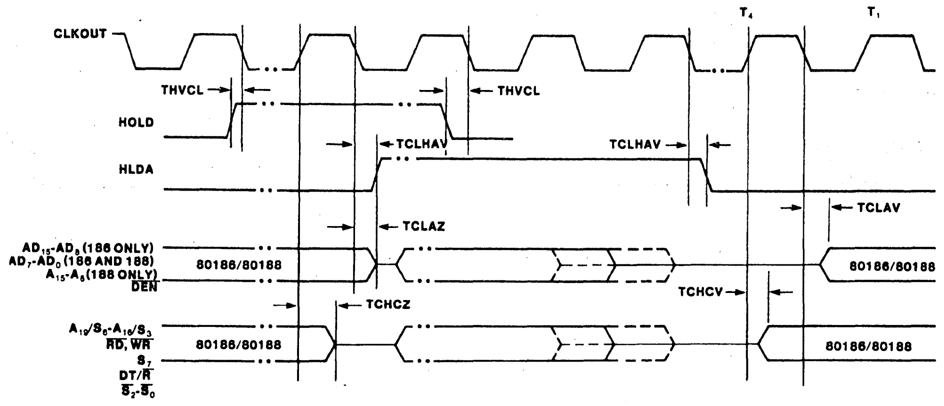
HOLD-HLDA TIMING



210469-21



210469-22

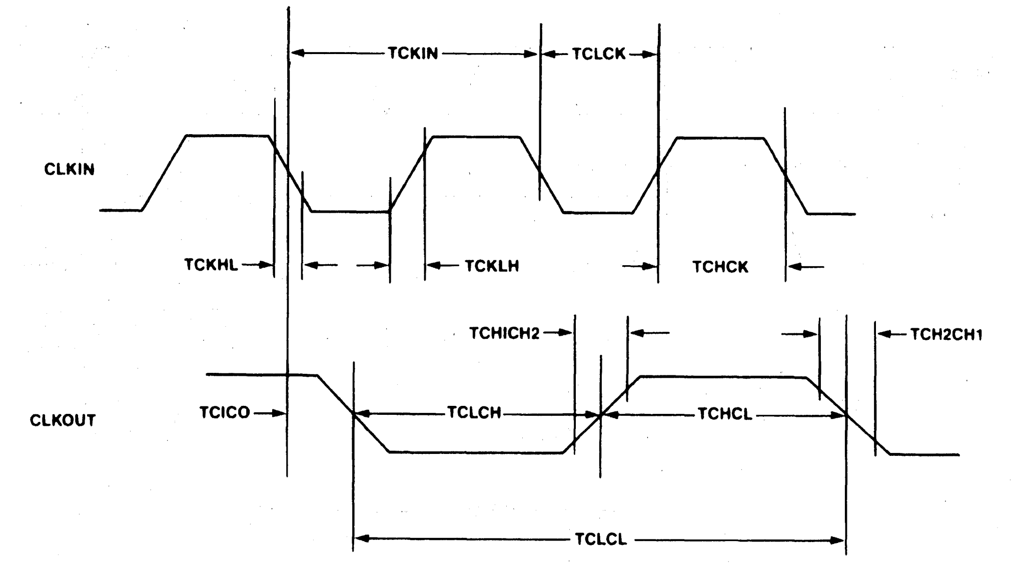


210469-23

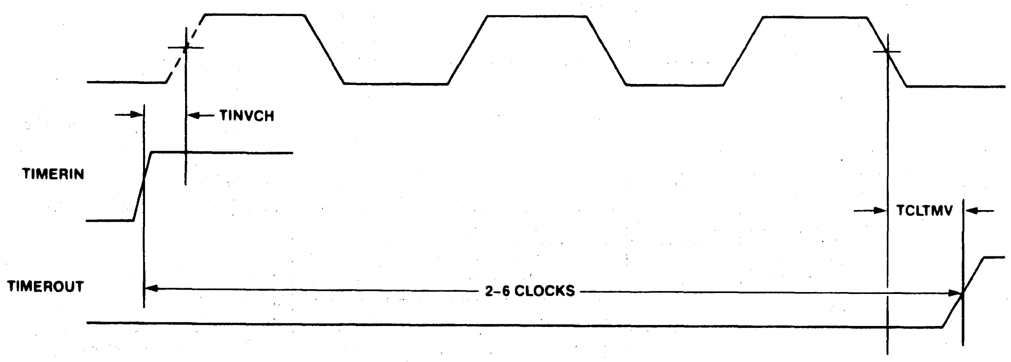
Figure 14

80186/80188 PROBE WAVEFORMS (Continued)

TIMER



210469-24



210469-25

Figure 15

I²ICE™ SYSTEM 80286 PROBE HIGHLIGHTS

Both 6 MHz and 8 MHz probes are available. Each probe has the following features:

- Supports real and protected mode (software).
- Includes an object code loader for both 8086 and 80286 object files.
- Supports multiprocessing (with coprocessor and with the 80287 processor extension).
- Supports local descriptor tables (LDTs).
- Provides full 24-bit address mapping (with optional 16K granularity).
- Provides the capability to read/write normally invisible portions of segment and table registers.
- Supports multitasking.
- Does not slip on breakpoints.
- DMA (Hold/Hold Acknowledge) is supported in both emulation and interrogation modes.

Table 18. I²ICE™ 80286 User Interface—D.C. Characteristics: 6 MHz Probe/8 MHz Probe

Pin Name	Input Voltage		Output Voltage		Input Current		Output Current	
	Max V_{IL} V	Min V_{IH} V	Max V_{OL} V	Min V_{OH} V	Max I_{IL} mA	Max I_{IH} mA	Max I_{OL} mA	Max I_{OH} mA
A0–A23			0.6/0.6	2.2/2.2			12/12	−3/−3
D0–D15	0.6/0.6	2.2/2.2	0.6/0.6	2.2/2.2	−0.1/−0.7	0.02/0.07	12/20	−3/−3
$\overline{S0}$, $\overline{S1}$			0.75/0.75	2.2*/2.2*			64/64	−3.4/−3.4
M/$\overline{IO}$			0.75/0.75	2.2/2.2			64/64	−3.0/−3.0
$\overline{LOCK}$			0.75/0.75	2.2/2.2			64/64	−3.0/−3.0
COD/$\overline{INTA}$			0.75/0.75	2.2/2.2			64/64	−1/−3
$\overline{BHE}$			0.75/0.75	2.2/2.2			64/64	−1/−3
$\overline{ERROR}$	0.6/0.6	2.2/2.2			−2.15/−3.7	0.05/0.04		
$\overline{BUSY}$	0.6/0.6	2.2/2.2			−0.4/−0.7	0.05/0.02		
$\overline{PEACK}$			0.75/0.75	2.5/2.2			64/64	−1/−3
HLDA			0.75/0.75	2.5/2.2			64/64	−1/−3
HOLD	0.5/0.5	2.3/2.3			−0.4/−0.7	0.05/0.02		
PEREQ	0.6/0.6	2.2/2.2			−1.6/−0.7	0.02/0.02		
INTR	0.5/0.5	2.3/2.3			−1.6/−0.7	0.02/0.02		
NMI	0.6/0.6	2.2/2.2			−1.6/−0.7	0.02/0.02		
$\overline{CLK}$	0.6/0.6	2.2/2.2			−2.0/−0.7	0.07/0.02		
$\overline{READY}$	0.5/0.5	2.2/2.2			−3.6/−2.4	0.09/0.06		

NOTES:
*$\overline{S0}$ and $\overline{S1}$ have 5.1K pullup resistors.
1. DC characteristics are given in the form *m/n,* where *m* is the characteristic for the 6 MHz 80286 probe and *n* is the characteristic for the 8 MHz probe. Negative currents (−) are defined as currents flowing out of a terminal, and positive currents are defined as currents flowing into a terminal.
2. The 80286 chip specification indicates that the chip has an output drive capacity of I_{OH} = − 400 μA and I_{OL} = 2.0 mA; the chip's input and 3-state loading specification is ±10 μA. As can be seen from the above table, the 80286 probe has a greater output drive capacity and presents higher input loading than the 80286 chip.
3. The 80286 probe does not draw any current from the user V_{CC}.

Capacitive Loading—80286 Probe

- The 80286 probe presents the user system with a maximum capacitive load of 80 pF (100 pF for $\overline{\text{READY}}$, HLDA, $\overline{\text{LOCK}}$; 130 pF for HLD, $\overline{\text{ERROR}}$, PEREQ, NMI RESET, INT, $\overline{\text{BUSY}}$; and 160 pF for $\overline{\text{BHE}}$).
- All 80286 probe outputs are capable of driving 0 pF while meeting all the probe's timing specifications. The 80286 probe will drive larger capacitive loads, but with possible performance degradation. Derate the timing specifications by 0.04 ns/pF corresponding to input capacitance of the user system.

A.C. CHARACTERISTICS FOR THE I²ICE™ 80286 SYSTEM PROBE

Table 19 provides timing information on the 80286 probe. Figures 16 through 19 define the timing symbols.

Table 19. Calculated Worst Case Timing Information

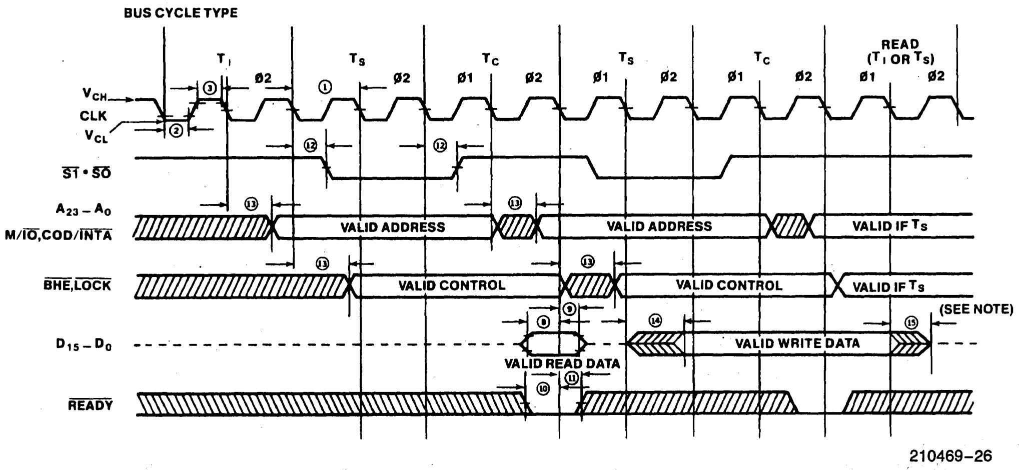
Symbol	Parameter	6 MHz Probe		8 MHz Probe	
		Min. ns	Max. ns	Min. ns	Max. ns
t1	System Clock Period	83	250	62	250
t2	System Clock Low Time	20	225	15	225
t3	System Clock High Time	25	230	225	235
t17	System Clock Rise Time		10		10
t18	System Clock Fall Time		10		10
t4	NMI, ITR, PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ Setup Time	27		20	
t4	HOLD Setup Time	40 (30)		28 (20)	
t5	NMI, INTR, PEREQ Hold Time	30		20	
t5	$\overline{\text{ERROR}}$ Hold Time	34 (30)		21 (20)	
t5	$\overline{\text{BUSY}}$ Hold Time	34 (30)		20	
t5	HOLD Hold Time	28		20	
t6	RESET Setup Time	12		28	
t7	RESET Hold Time	13 (5)		13 (5)	
t8	Read Data in Setup Time	20		11 (10)	
t9	Read Data in Hold Time	10 (8)		10 (8)	
t10	$\overline{\text{READY}}$ Setup Time	50		38	
t11	$\overline{\text{READY}}$ Hold Time	35		25	
t12	STATUS Valid Delay	10 (1)	55	1	44 (40)
t13	Address Valid Delay	10 (1)	80	1	65 (60)
t14	Write Data Valid Delay	9 (0)	65	0	59 (50)
t15	Address/STATUS/Data Float Delay (See Tables 20–23)	—	—	—	—
t15	Write Data Float Delay	10 (0)	34	0	50
t16	HLDA Valid Delay	9 (0)	80	0	60
t23	$\overline{\text{PEACK}}$ Valid Delay	10 (1)	80 (55)	1	48 (40)

Numbers followed by parentheses deviate from the 80286 chip specification; the 1985 *Microsystem Components Handbook* chip specification timing is given in the parentheses.

NOTE:
1. The symbols t1, t2, t3 ... are references for the circled numbers on Figures 16, 17, 18, 19, and 24. For example to find t14 "Write Data Valid Delay" in Figure 16, look for a circled 14 on the figure.

80286 PROBE WAVEFORMS

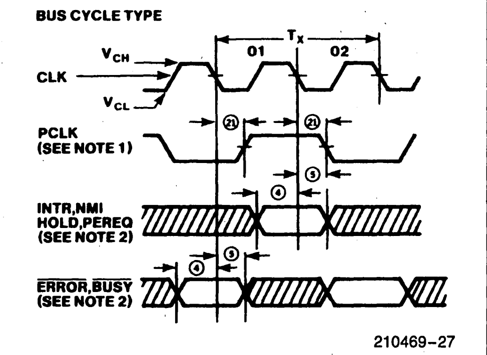
MAJOR CYCLE TIMING



NOTE:
Write data hold time can be increased by mapping memory to high-speed (HS) or optional high-speed (OHS) memory and using the WAITSTATE command to specify more waitstates than the number requested by the target system READY̅. Write data hold time can be extended even further by mapping memory to MULTIBUS memory. Mapping I/O to the I²ICE system (using the ICE option with the MAPIO command) also causes write data hold time to increase for I/O write cycles.

Figure 16

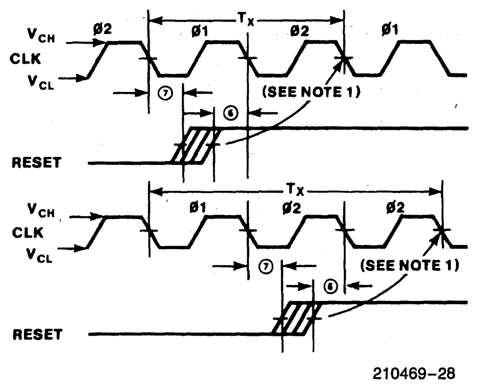
80286 ASYNCHRONOUS INPUT SIGNAL TIMING



NOTES:
1. PCLK indicates which processor cycle phase will occur on the next CLK. PCLK may not indicate the correct phase until the first bus cycle is performed.
2. These inputs are asynchronous. The setup and hold times shown assure recognition for testing purposes.

Figure 17

80286 RESET INPUT TIMING AND SUBSEQUENT PROCESSOR CYCLE PHASE

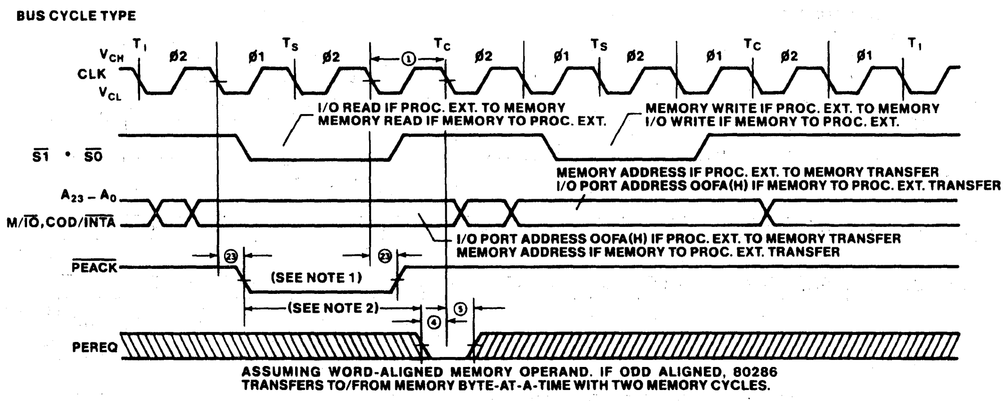


NOTE:
1. When RESET meets the setup time shown, the next CLK will start or repeat 02 of a processor cycle.

Figure 18

80286 PROBE WAVEFORMS (Continued)

80286 PEREQ/PEACK TIMING FOR ONE TRANSFER ONLY


BUS CYCLE TYPE
T_I
T_S
T_C
T_S
T_C
T_I
V_CH
CLK
V_CL
ø2
ø1
ø2
①
ø2
ø1
ø2
ø1
ø2
ø1
I/O READ IF PROC. EXT. TO MEMORY
MEMORY READ IF MEMORY TO PROC. EXT.
MEMORY WRITE IF PROC. EXT. TO MEMORY
I/O WRITE IF MEMORY TO PROC. EXT.
S1 • S0
MEMORY ADDRESS IF PROC. EXT. TO MEMORY TRANSFER
I/O PORT ADDRESS OOFA(H) IF MEMORY TO PROC. EXT. TRANSFER
A23 – A0
M/IO, COD/INTA
I/O PORT ADDRESS OOFA(H) IF PROC. EXT. TO MEMORY TRANSFER
MEMORY ADDRESS IF MEMORY TO PROC. EXT. TRANSFER
㉓
㉓
PEACK
(SEE NOTE 1)
(SEE NOTE 2)
④
⑤
PEREQ
ASSUMING WORD-ALIGNED MEMORY OPERAND. IF ODD ALIGNED, 80286 TRANSFERS TO/FROM MEMORY BYTE-AT-A-TIME WITH TWO MEMORY CYCLES.

210469–29

NOTES:
1. PEACK always active during the first bus operation of a processor extension data operant transfer sequence. The first bus operation will be either a memory read at operand address or I/O at port address 00FA(H).
2. To prevent a second processor extension data operand transfer, the worst case maximum time (shown above) is: 3 × ① − ⑪ max. − 4 min. The actual, configuration-dependent, maximum time is: 3 × ① − ⑪ max. − ④ min + A × 2 × ①. A is the number of extra T_C states added to either the first or second bus operation of the processor extension data operand transfer sequence.

Figure 19

Interrupt Acknowledge Sequence Timing

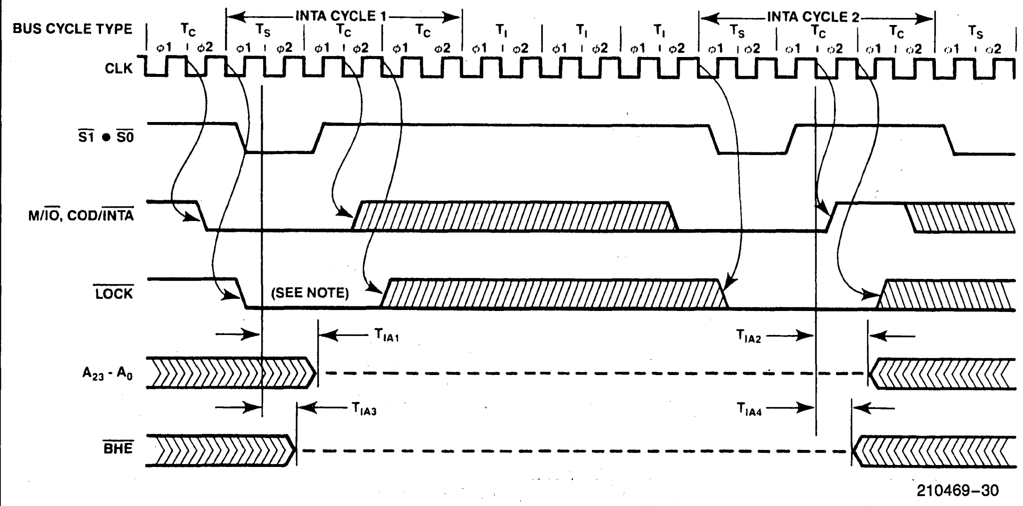
Figure 20 shows an interrupt acknowledge sequence for the 80286 probe. Table 20 provides timing information.

Table 20. Interrupt Acknowledge Timing

Symbol	Definition	6 MHz Probe		8 MHz Probe	
		Min. ns	Max. ns	Min. ns	Max. ns
T_{IA1}	Address Float Delay	14	59	19	60
T_{IA2}	Address Valid Delay	23	79	23	68
T_{IA3}	$\overline{BHE}$ Float Delay	14	56	18	51
T_{IA4}	$\overline{BHE}$ Valid Delay	18	68	17	51

80286 PROBE WAVEFORMS (Continued)

INTERRUPT ACKNOWLEDGE SEQUENCE TIMING



NOTE:
See pages 4–33 in the 1985 *Microsystem Components Handbook.*

Figure 20

80286 PROBE WAVEFORMS (Continued)

Reset Differences Between the 80286 and the I²ICE™ System 80286 Probe

There is a two-clock cycle delay that the I²ICE system 80286 probe adds to the RESET that it receives from the target system. This delay affects the initial 80286 pin state during reset as seen in the target system. A diagram showing the normal 80286 pin state during RESET can be found in the iAPS-286/10 data sheet. The effects of the two-clock delay and other emulator pin-state differences (as a function of target system RESET) are shown in Table 22. It should be noted that target system RESET has no effect if RSTEN is FALSE. The timing symbols used in Table 22, and Figures 21 and 22, are defined in Table 21.

Table 21. Timing Signal Definitions for Table 22 and Figures 21 and 22

Timing Signal	Definition
T_{R1}	User RESET Setup Time to User CLOCK Falling Edge.
T_{R2}	User RESET Hold Time after User CLOCK Falling Edge.
T_{R3}	ADDRESS Float Delay Due to RESET High. (1)
T_{R4}	ADDRESS Active after RESET Goes Low. (1)
T_{R5}	$\overline{\text{BHE}}$ Float Delay Due to RESET High. (1)
T_{R6}	$\overline{\text{BHE}}$ Active Delay after RESET Low. (1)
T_{R7}	M/$\overline{\text{IO}}$, $\overline{\text{S0}}$ and $\overline{\text{S1}}$ Float Delay after RESET High (1, 2)
T_{R8}	M/$\overline{\text{IO}}$, $\overline{\text{S0}}$ and $\overline{\text{S1}}$ Active Delay after RESET Low (1, 2)
T_{R9}	DATA Float Delay after RESET High. (1)
T_{R10}	$\overline{\text{LOCK}}$ High after RESET High. (1)
T_{R11}	COD/$\overline{\text{INTA}}$ Low after RESET High. (1)
T_{R12}	$\overline{\text{PEACK}}$ High after RESET High. (1)

NOTES:
1. RESET must meet setup and hold requirements T_{R1} and T_{R2}.
2. $\overline{\text{S0}}$ and $\overline{\text{S1}}$ are pulled up with 5.1K resistors on the emulator. The float delay in this case is the delay until status is no longer actively driven by the emulator. The active delay is the time until status is actively driven high.

Table 22. Emulator Pin State Differences Due to RESET

Symbol	Calculated Worst-Case				Number of CLOCK Edges After User RESET Goes High or Low
	6 MHz Probe		8 MHz Probe		
	Min. ns	Max. ns	Min. ns	Max. ns	
T_{R1}	12		23		Not Applicable
T_{R2}	13		13		Not Applicable
T_{R3}	31	97	35	85	2 (high)
T_{R4}	32	102	41	95	2 (low)
T_{R5}	32	109	35	79	2 (high)
T_{R6}	33	101	35	78	2 (low)
T_{R7}	26	83	32	72	2 (high)
T_{R8}	26	85	32	71	2 (low)—See Note 2 at Bottom of Table 21.
T_{R9}	19	62	28	66	2 (high)
T_{R10}	9	72	10	61	8 (high)
T_{R11}	10	76	10	61	9 (high)
T_{R12}	10	76	10	48	8 (high)

80286 PROBE WAVEFORMS (Continued)

Figure 21 and 22 show the RESET pin timing differences between the I²ICE 80286 probe and 80286 chip.

PIN TIMING DIFFERENCES DUE TO RESET FOR T_{R1} TO T_R

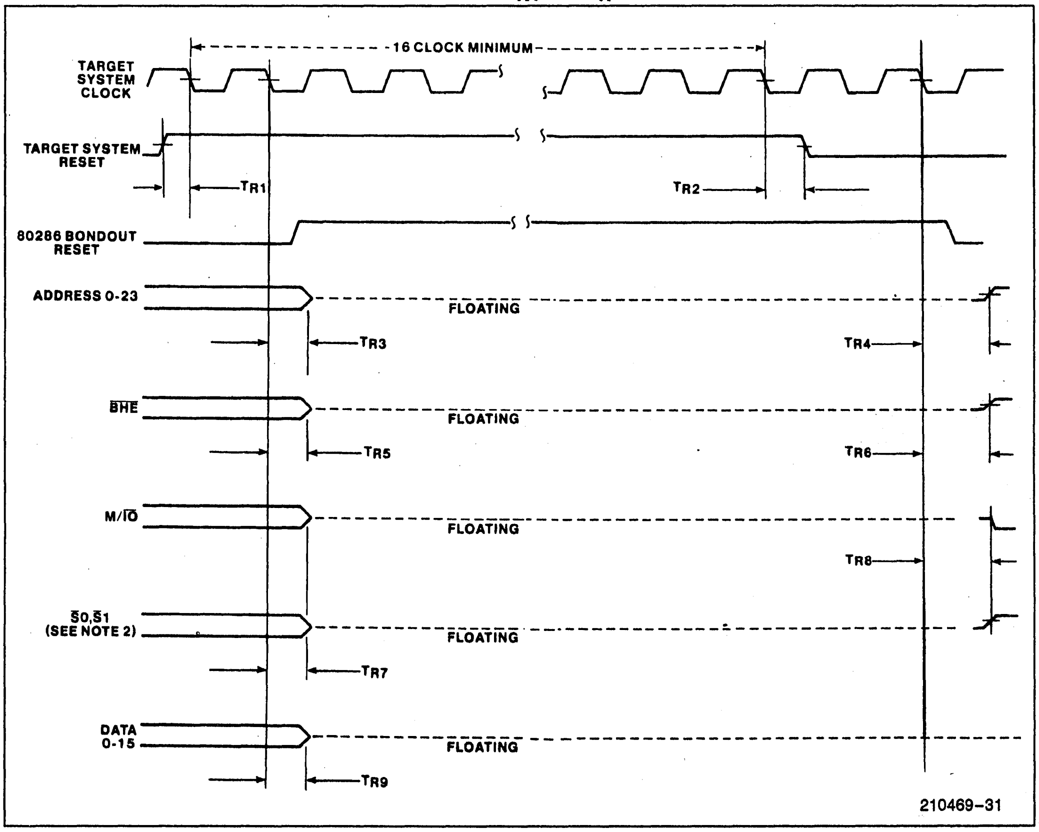


Figure 21

80286 PROBE WAVEFORMS (Continued)

PIN TIMING DIFFERENCES DUE TO RESET FOR T_{R10} TO T_{R12}

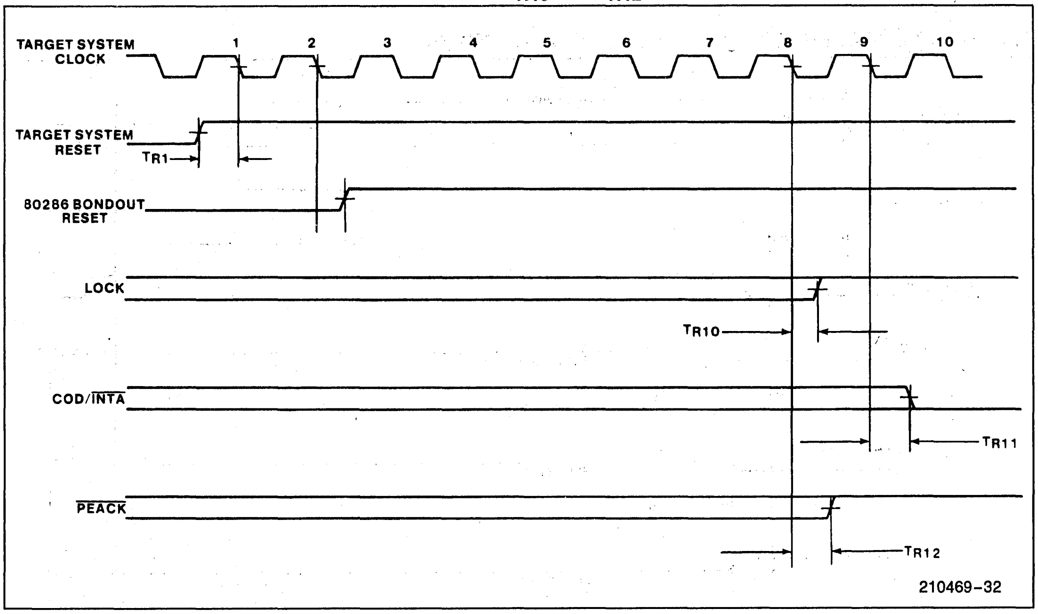


Figure 22

HOLD/HLDA Differences Between the 80286 and the I2ICE™ System 80286 Probe

There are differences in the pin timing parameters of the I2ICE 80286 probe and the 80286 as a function of HLDA. A diagram showing the pin timing while exiting and entering HOLD can be found in the iAPX-286/10 data sheet. There are two I2ICE pseudo-variables that control when a HOLD request will be honored. When COENAB is FALSE, HOLD requests from the target system are disabled. When COENAB is TRUE, the pseudo-variable CPMODE controls when HOLD is recognized. If CPMODE equals 1, HOLD will only be recognized when the probe is in emulation. If CPMODE is equal to 2, HOLD will always be recognized, even when the I2ICE system is in interrogation mode (not emulating). Table 24 gives the specification for the I2ICE 80286 pin timing as a function of HLDA. The timing signals used in Table 24, and Figures 23 and 24, are defined in Table 23.

Table 23. Timing Signal Definitions for Table 22 and Figures 23 and 24

Timing Signal	Definition
T_{H1}	$\overline{\text{LOCK}}$ Float Delay from HLDA High.(2)
T_{H2}	$\overline{\text{LOCK}}$ Active Delay from HLDA Low.(2)
T_{H3}	$\overline{\text{BHE}}$ Float Delay from HLDA High.
T_{H4}	$\overline{\text{BHE}}$ Active Delay from HLDA Low.
T_{H5}	M/$\overline{\text{IO}}$ Float Delay from HLDA High.
T_{H6}	M/$\overline{\text{IO}}$ Active Delay from HLDA Low.
T_{H7}	COD/$\overline{\text{INTA}}$ Float Delay from HLDA High.
T_{H8}	COD/$\overline{\text{INTA}}$ Active Delay from HLDA Low.
T_{H9}	ADDRESS 0–23 Float Delay from HLDA High.
T_{H10}	ADDRESS 0–23 Active Delay from HLDA Low.
T_{H11}	DATA 0–15 Float Delay from READY of Last Write Cycle before HLDA Goes HIgh. (80286 t15)
T_{H12}	DATA 0–15 Active Delay from HLDA Low. (80286 t14)
T_{H13}	$\overline{\text{S0}}$ and $\overline{\text{S1}}$ Float Delay from HLDA High.(1)
T_{H14}	$\overline{\text{S0}}$ and $\overline{\text{S1}}$ Active Delay from HLDA Low.(1)
T_{H15}	$\overline{\text{PEACK}}$ Float Delay from HLDA High.
T_{H16}	$\overline{\text{PEACK}}$ Active Delay from HLDA Low.
T_{H17}	HLDA High to Low Valid Delay.(3)
T_{H18}	HLDA Low to High Valid Delay.(3)

NOTES:
1. $\overline{\text{S0}}$ and $\overline{\text{S1}}$ are pulled up with 5.1K resistors on the emulator. The float delay in this case is the delay until status is no longer actively driven by the emulator. The active delay is the time until status is actively driven high.
2. The specified active delay indicates the time from the falling edge of target system clock until the I2ICE actively drives the signal. In some cases, the signal may not become valid until after the valid delay specified for normal (no HLDA) bus cycles. One such case is $\overline{\text{LOCK}}$.
3. All float delays and active delays (except T_{H11} and T_{H12}) are due to a combinational delay from HLDA.

Table 24. Pin Timing Differences Due to HOLD

Symbol	Calculated Worst-Case			
	6 MHz Probe		8 MHz Probe	
	Min ns	Max ns	Min ns	Max ns
T_{H1}	19	112	20	81
T_{H2}	19	112	19	78
T_{H3}	21	120	22	85
T_{H4}	21	115	21	82
T_{H5}	18	113	17	78
T_{H6}	20	113	17	75
T_{H7}	17	106	17	75
T_{H8}	19	106	16	72
T_{H9}	21	119	22	91
T_{H10}	22	127	21	82
T_{H11}	*	*	*	*
T_{H12}	*	*	*	*
T_{H13}	18	113	17	78
T_{H14}	19	113	17	75
T_{H15}	17	106	17	75
T_{H16}	17	105	16	72
T_{H17}	9	80	0	60
T_{H18}	9	80	0	60

*See Figures 23 and 24.

80286 PROBE WAVEFORMS (Continued)

ENTERING AND EXITING HOLD

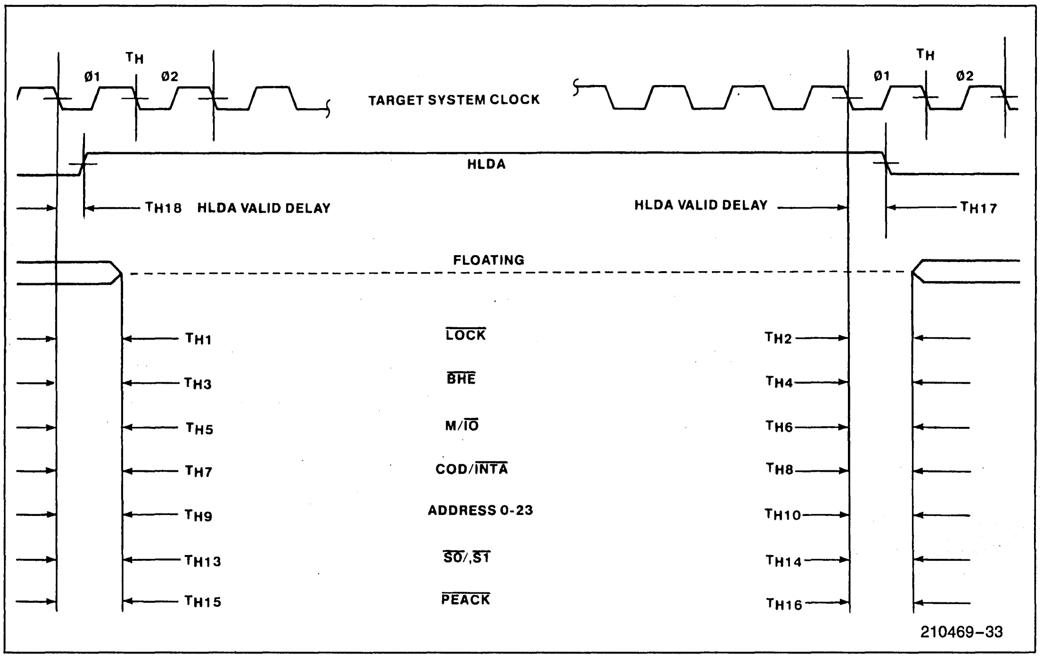
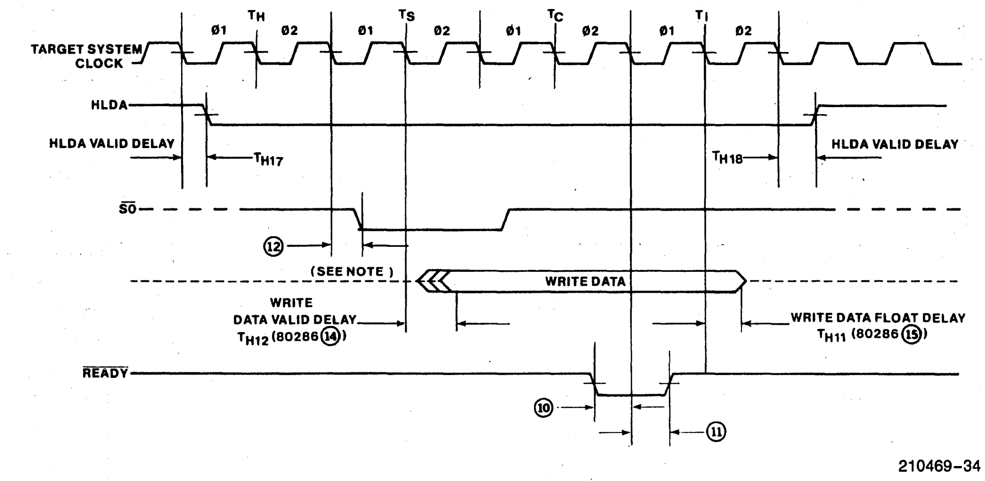


Figure 23

80286 PROBE WAVEFORMS (Continued)

EXITING AND ENTERING HOLD (Continued)



NOTE:
1. The data bus will remain 3-state OFF if a read cycle is performed.

Figure 24

Available Documentation

122143	*AEDIT™ Text Editor User's Guide*
121790	*PSCOPE 86 High-Level Program Debugger User's Guide*
166298	*I²ICE™ System User's Guide*
166302	*I²ICE™ System Reference Manual*
166305	Installation Supplement for *I²ICE™ System User's Guide* for Intel hosts
166306	Installation Supplement for *I²ICE™ System User's Guide* for IBM PC hosts
163256	*iLTA User's Guide*
163257	*iLTA Reference Manual*
163258	*iLTA Learner's Guide*
210350	*PSCOPE 86 data sheet*
230839	*iLTA data sheet*

ORDERING INFORMATION

Order Code	Description: Kits for Series III Host
III010KITB	I²ICE system 8086/8088 support kit for Series III host. Includes III086A902B (8086/8088 probe and software), III515 (instrumentation chassis and emulation base module), and III520B952B (host interface module and host software).
III110KITB	I²ICE system 80186/80188 support kit for Series III host. Includes III186A912B (80186/80188 probe and software) III515 (instrumentation chassis and emulation base module), and III520B952B (host interface module and host software).
III210KITB	I²ICE system 6 MHz 80286 support kit for Series III host. Includes III286A922B (6 MHz 80286 probe and software), III515 (instrumentation chassis and emulation base module), and III520B952B (host interface module and host software).

Order Code	Description: Kits for Series III Host (Continued)
IIIOLAKITB	I2ICE system iLTA-OHS support kit for Series III host. Includes III707 (OHS memory board), III515 (instrumentation chassis and emulation base module), III810A982B (logic timing analyzer and iLTA host software), and III520B952B (host interface module and host software).
III811KITB	Series III iLTA kit, includes III810A982B (iLTA hardware and software) and III820 (IOC board)

Order Code	Description: Probes and Interface Module with Required Software for Series III Host
III086A902B	I2ICE system 8086/8088 emulation personality module (probe) and probe software (8-in. single-density and double-density disks).
III186A912B	I2ICE system 80186/80188 emulation personality module (probe) and probe software (8-in. single-density and double-density disks).
III286A922B	I2ICE system 6 MHz 80286 emulation personality module (probe) and probe software (8-in. single-density and double-density disks).
III286B922B	I2ICE system 8 Mhz 80286 emulation personality module (probe) and probe software (8-in. single-density and double-density disks).
III520B952B	Series III host interface module; includes host interface board, cables, and I2ICE system host software (8-in. single-density and double-density disks).

Order Code	Description: Kits for Series IV Host
III010KITC	I2ICE system 8086/8088 support kit for Series IV host. Includes III186A903C (8086/8088 probe and software), III515 (instrumentation chassis and emulation base module), and III520B953C (host interface module and host software).
III110KITC	I2ICE system 80186/80188 support kit for Series IV host. Includes III186A913C (80186/80188 probe and software), III515 (instrumentation chassis and emulation base module), and III520B953C (host interface module and host software).
III210KITC	I2ICE system 6 MHz 80286 support kit for Series IV host. Includes III286A923C (6 MHz 80286 probe and software), III515 (instrumentation chassis and emulation base module), and III520B953C (host interface module and host software).
IIIOLAKITC	I2ICE system iLTA-OHS software kit for Series IV host. Includes III707 (OHS memory board), III515 (instrumentation chassis and emulation base module), and III810A983C (logic timing analyzer and iLTA host software), and III520B953C (host interface module and host software).

Order Code	Description: Probes and Interface Module with Required Software for Series IV Host
III086A903C	I2ICE system 8086/8088 emulation personality module (probe) and probe software (5¼-in. double-density disk).
III186A913C	I2ICE system 80186/80188 emulation personality module (probe) and probe software (5¼-in. double-density disk).
III286A923C	I2ICE system 6 MHz 80286 emulation personality module (probe) and probe software (5¼-in. double-density disk).

Order Code	Description: Probes and Interface Module with Required Software for Series IV Host (Continued)
III286B923C	I^{2}ICE system 8 MHz 80286 emulation personality module (probe) and probe software (5¼-in. double-density disk).
III520B953C	Series IV host interface module; includes the host interface board, cables, and I^{2}ICE system host software (5¼-in. double-density disks).

Order Code	Description: Kits for IBM PC Host
III010KITD	I^{2}ICE system 8086/8088 support kit for IBM PC host. Includes III086A904D (8086/8088 probe and software), III515 (instrumentation chassis and emulation base module), and III520AT954D (host interface module and host software).
III110KITD	I^{2}ICE system 80186/80188 support kit for IBM PC host. Includes III186A914D (80186/80188 probe and software), III515 (instrumentation chassis and emulation base module), and III520AT954D (host interface module and host software).
III210KITD	I^{2}ICE system 6 MHz 80286 support kit for IBM PC host. Includes III286A924D (6 MHz 80286 probe and software), III515 (instrumentation chassis and emulation base module), and III520AT954D (host interface module and host software).

Order Code	Description: Probes and Interface Module with Required Software for IBM PC Host
III086A904D	I^{2}ICE system 8086/8088 emulation personality module (probe) and probe software for PC-DOS (5¼-in. double-density disks).
III186A914D	I^{2}ICE system 80186/80188 emulation personality module (probe) and probe software for PC-DOS (5¼-in. double-density disk).
III286A924D	I^{2}ICE system 6 MHz 80286 emulation personality module (probe) and probe software for PC-DOS (5¼-in. double-density disk).
III286A924D	I^{2}ICE system 8 MHz 80286 emulation personality module (probe) and probe software for PC-DOS (5¼-in. double-density disk).
III520AT954D	IBM PC host interface module; includes the host interface board, cable, and I^{2}ICE system host software for PC-DOS (5¼-in. double-density disks).

Order Code	Description: I^{2}ICE System Instrumentation Chassis and Optional High Speed Memory for Series III and IV and IBM PC Hosts
III515	I^{2}ICE system instrumentation chassis and emulation base module; includes the instrumentation chassis, emulation base module with memory map I/O board, break/trace board, emulator base, and emulation clips assembly.
III707	I^{2}ICE system optional high speed (OHS) memory board, 128K bytes.

Order Code	Description: iLTA Logic Timing Analyzer, Cables, and Accessories
III810A982B	I^{2}ICE system logic timing analyzer and software for operation with a Series III host (8-in. single-density and double-density disks).
III810A983C	I^{2}ICE system logic timing analyzer and software for operation with a Series IV host (5¼-in. double-density disks).

Order Code	Description: iLTA Logic Timing Analyzer, Cables, and Accessories (Continued)
III530	Cable, host-to-chassis, 3 m (10 ft.) [only for use with Model 800 hosts].
III531*	Cable, host-to-chassis, 12.8 m (42 ft.) [only for use with Model 800 hosts].
III531A*	Cable, host-to-chassis (for Series III and Series IV hosts); 12.2 m (40 ft).
III532A	Cable set, inter-chassis; 0.6 m (2 ft).
III533A*	Cable set, inter-chassis; 3 m (10 ft).
III621	I²ICE system emulator clips assembly.
III815	I²ICE system micro hook set (40).
III820	IOC board upgrade for iLTA on Series III development system.

*Do not use III531 or III531A with III533A because total cable length must be less than 15.2 m (50 ft.)

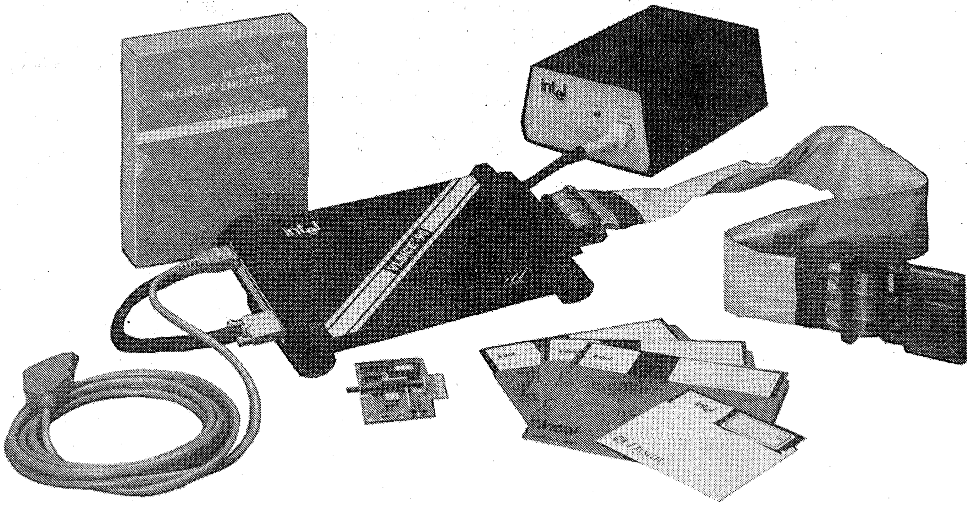
Order Code	Description: Extra I²ICE System Software
902B	I²ICE system 8086/8088 probe software for Series III (8-in. single-density and double-density disks).
903C	I²ICE system 8086/8088 probe software for Series IV (5¼-in. double-density disk).
904D	I²ICE system 8086/8088 probe software for PC-DOS (5¼-in. double-density disk).
912B	I²ICE system 80186/80188 probe software for Series III (8-in. single-density and double-density disks).
913C	I²ICE system 80186/80188 probe software for Series IV (5¼-in. double-density disk).
914D	I²ICE system 80186/80188 probe software for PC-DOS (5¼-in. double-density disk).
922B	I²ICE system 80286 probe software for Series III (8-in. single-density and double-density disks).
923C	I²ICE system 80286 probe software for Series IV (5¼-in. double-density disk).
924D	I²ICE system 80286 probe software for PC-DOS (5¼-in. double-density disk).
952B	I²ICE system host software for Series III (8-in. single-density and double-density disks).
953C	I²ICE system host software for Series IV (5¼-in. double-density disks).
954D	I²ICE system host software for PC-DOS (5¼-in. double-density disks).
982B	I²ICE system iLTA software for Series III (8-in. single-density and double-density disks).
983C	I²ICE system iLTA software for Series IV (5¼-in. double-density disks).

Order Code	Description: PSCOPE on IBM PC-DOS
iPSC86DOS	PSCOPE software that runs on IBM PC-DOS.

VLSiCE™-96P IN-CIRCUIT EMULATOR FOR THE MCS®-96 FAMILY OF MICROCONTROLLERS

- **Precise Real-Time Emulation of the MCS®-96 Family of Components**
- **64K of Mappable Memory for Early Software Debug and (EP)ROM Simulation**
- **A 4K-Entry Trace Buffer for Storing a Real-Time Execution History, Including Both Code and Data Flows**
- **Multistate Break and Trace Qualified on Execution Addresses, Data Addresses, and Values (Both External and Internal RAM), Opcodes and Selected PSW Flags**
- **Fast Break and Dynamic Trace**
- **Shadow Registers Allow Reading Many 8096 Write-Only and Writing Many Read-Only Registers**
- **Symbolic Debugging Allows Accesses to Memory Locations and Program Variables (Including Dynamic Variables) Using Program-Defined Names**
- **Equipped with the Integrated Command Directory (ICD) Which Provides**
  - **—An On-Line Help File**
  - **—A Dynamic Syntax Menu**
  - **—Dynamic Command-Entry Error Checking**
- **Serially Linked to Intel Series III/IV Hosts or IBM* PC-XT and AT**

The VLSiCE™-96P In-Circuit Emulator is a debugging and test tool that is used for development of the hardware and software of a prototype system based on the MCS®-96 family of microcontrollers.



280140-1

*IBM is a trademark of International Business Machines.

November 1986
Order Number: 280140-003

INTRODUCTION

The VLSiCE-96P emulator allows hardware and software development of a design project to proceed simultaneously. With the VLSiCE-96P emulator, prototype hardware can be added to the system as it is designed and software can be developed prior to the completion of the hardware prototype. Software and hardware integration then occurs while the product is being developed.

The VLSiCE-96P emulator assists four stages of development:

- Software debugging
- Hardware development
- System integration
- System test

Software Debugging

The VLSiCE-96P emulator can be operated without being connected to the user's prototype or before any of the user's hardware is available. In this stand-alone mode, the VLSiCE-96P emulator can be used to facilitate program development.

Hardware Debugging

The VLSiCE-96P emulator's precise characteristics that match the controller and full-speed operation make it a valuable tool for debugging hardware, including time-critical serial port and timer interfaces.

System Integration

Integration of software and hardware can begin when the microcontroller socket is connected to any functional element of the user system hardware. As each section of the user's hardware is completed, it can be added to the prototype. Thus, each section of the hardware and software can be system tested in real-time operation as it becomes available.

System Test

When the prototype is complete, it is tested with the final version of the system software. The VLSiCE-96P emulator is then used for real-time emulation of the microcontroller to debug the system as a completed unit.

The final product verification test may be performed using the ROM or EPROM version of the microcontroller. Thus, the VLSiCE-96P emulator provides the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

PHYSICAL DESCRIPTION

The VLSiCE-96P emulator consists of the following components (see Figure 1):

- Power supply
- AC and DC power cables
- Serial cable (host-specific)
- Controller pod
- User cable assembly (consisting of the user cable and processor module)
- Crystal power accessory (CPA)
- Multi-synchronous accessory (MSA)
- 48-pin DIP adaptor
  68-pin PGA adaptor
  68-pin PLCC adaptor (optional)
- Software (includes the VLSiCE-96P emulator software, diagnostic software, and tutorial)

The power supply connects to the controller pod via the DC power cable. There are several voltage options available for the power supply determined by the settings of the switches on the back of the supply.

The controller pod contains 64K of ICE memory, a 4K-entry trace buffer, and circuitry which provides communication between the host and the emulator.

The serial cable connects the host system to the controller pod. The serial cable has electrical specifications similar to the RS-232C standard.

The processor module contains a special version of the Intel 8096 microcontroller, called the emulation processor. This chip performs real-time and single-step execution of a program's object code for debugging purposes and replaces the target system microcontroller.

The crystal power accessory (CPA) is a small detachable board that connects to the back of the controller pod and is used to run the VLSiCE-96P emulator in the stand-alone mode. It is also used when running customer confidence tests. In the stand-alone mode, the user plug on the user cable is connected through the 68-pin PGA adaptor to the CPA. The CPA supplies clock and power. Stand-alone mode is used to test and debug software prior to the availability of hardware.

The multi-synchronous accessory can be used to connect up to 21 multi-ICE compatible emulators to-

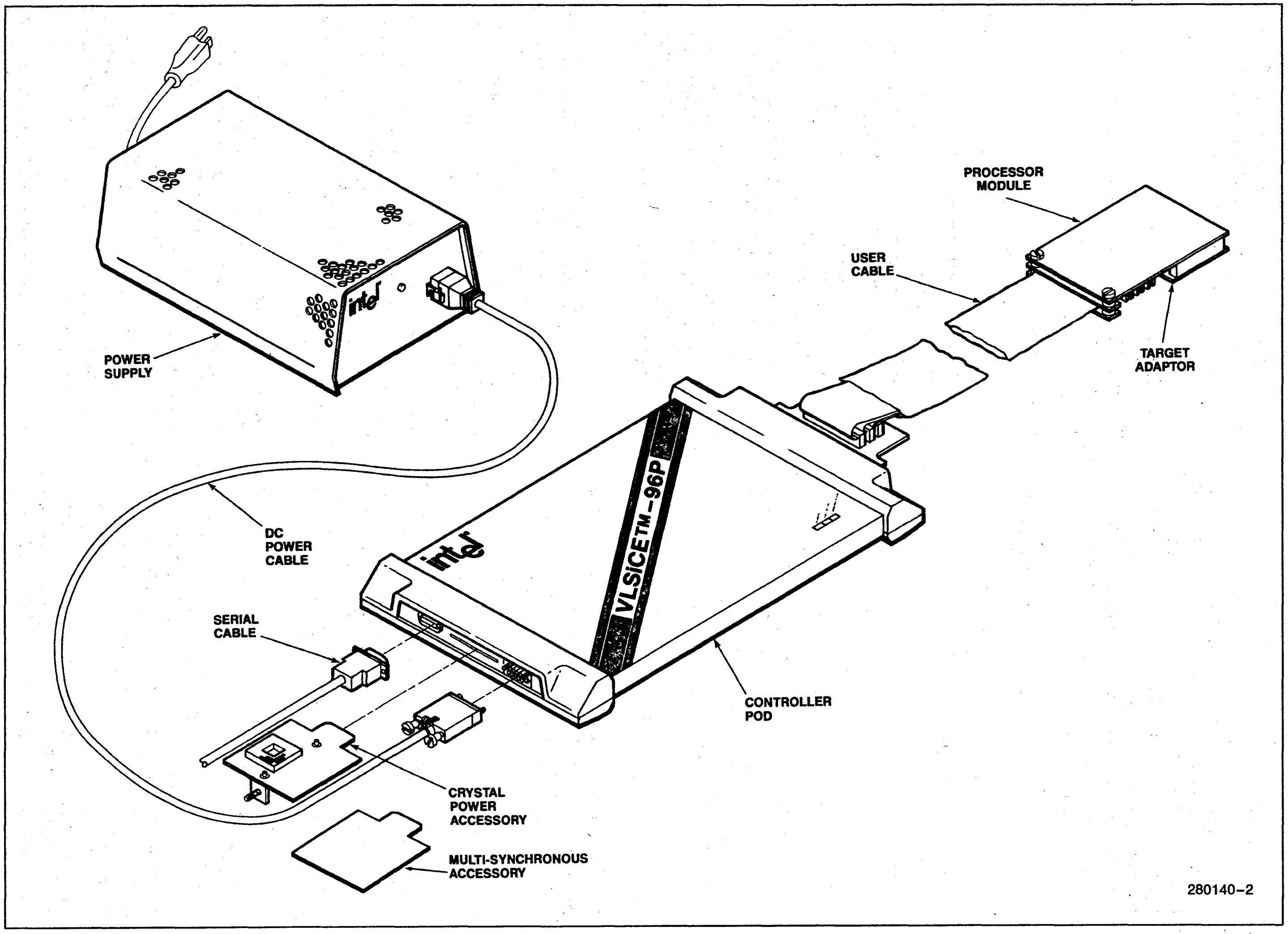


Figure 1. The VLSICE™-96P Emulator

gether for synchronous GO and BREAK emulation. It can also be used with other debug equipment such as logic analyzers and oscilloscopes.

Figure 1 shows a drawing of the VLSiCE-96P emulator.

VLSiCE software fully supports all mnemonics, object file formats, and symbolic references generated by Intel's ASM 96, PL/M 96, and C96.

The on-line tutorial is written in VLSiCE-96P command language. Thus, the user is able to interact with and use the VLSiCE-96P emulator while executing the tutorial.

A comprehensive set of documentation is included with the VLSiCE-96P emulator.

VLSiCE™-96P EMULATOR FEATURES

The VLSiCE-96P emulator has been created to assist a product designer in developing, debugging and testing designs incorporating the MCS-96 family of microcontrollers. The following are some of the VLSiCE-96P features:

Emulation

Emulation is the controlled execution of the prototype software in the prototype hardware or in an artificial hardware environment that duplicates the microcontroller of the prototype system. Emulation is a transparent process that happens in real-time. The execution of prototype software is facilitated through the VLSiCE-96P command language.

Memory Mapping

There is 64K of zero-waitstate, high-speed mappable memory available. This memory space can be mapped to either the user prototype system or to the on-board VLSiCE-96P memory space in 1K-byte blocks on 1K-byte boundaries. Mapping memory to the VLSiCE-96P system allows software development to proceed before prototype hardware is available. Memory mapping also gives the VLSiCE-96P emulator the capability to simulate the 8K-bytes of (EP)ROM on those versions of the chip for code verification and validation.

Memory Examination and Modification

The memory space for the MCS-96 component and its target hardware is accessible through the emulator. The VLSiCE-96P software allows the component's special function registers to be accessed mnemonically (e.g. AD RESULT, INT MASK). A significant benefit to the user is the ability of the VLSiCE-96P software to read many of the write-only registers (e.g. AD_COMMAND, PWM_CONTROL) and to write many of the read-only registers (e.g. AD_RESULT, SBUFRX). Data can be displayed or modified in several bases: hex, decimal, and binary, and in these formats: ASCII, real and integer. Program code can be disassembled and displayed with opcode mnemonics. It also can be modified with standard assembler statements.

Symbolic debugging is used to specify memory locations by their symbolic references. A symbolic reference is a procedure name, line number, or label in the user program that corresponds to a location. Using symbolics to reference program locations is a mnemonic way of accessing the program.

Some typical symbolic functions include:

- Changing or inspecting the value of a program variable by using its symbolic name, rather than the address of the memory location.
- Defining break and trace events using symbolic references.
- Referencing static variables, dynamic (stack-resident) variables, based variables, and record structures combining primitive data types. The primitive data types are ADDRESS, BOOLEAN, BYTE, CHAR (character), WORD, DWORD (double word), INTEGER, LONGINT, SHORTINT, and REAL.

The VLSiCE-96P system maintains a virtual symbol table for program symbols making it possible for the table to exist without fitting entirely into host RAM memory. The size of the virtual symbol table is constrained only by the capacity of the disk.

Breakpoint Specifications

Breakpoints allow halting of a user program in order to examine the effect of the program's execution on the user prototype system. Breakpoints can be defined as execution addresses, data addresses and data values (both external and **internal RAM**, opcodes, and selected bids of the PSW flag. These breaks can also be arranged to occur over a range of addresses. After a break the user program can resume execution from where it left off.

Trace Specifications

Tracing can be triggered with the same conditions set for breaking. The trace buffer is displayed as disassembled instructions, data fetches and stores, and with the timetag showing the relative time at

```
hlt> PRINT NEWEST 8 CYCLES
FRAME ADDRESS CODE         MNEMONIC OPERANDS      TIME

(1020) 300A C82A           PUSH 2A                  172 µs
        [002AH]=0087H(R) [0018H]=0028H(R)      [0018H]=0026H(W)
        [0026H]=0087H(W)

(1021) 300C 64322A         ADD 2A,32                175 µs
        [0032H]=0010H(R) [002AH]=0087H(R)      [002AH=0097H(W)

(1022) 300F 6975002A       SUB 2A,#75               180 µs
        [002AH]=0097H(R) [002AH]=0022H(W)
•
•
•
•
•
(1027) 3010 CC2A0002       POP 2A                   205 µs
        [0018H]=0026H(R) [0026H]=0087H(R)      [0018H=0028H(W)
        [002AH]=0087H(W)
```

Figure 2. The Trace Buffer Display

which the program executed each instruction. Figure 2 shows a trace display as a result of the PRINT command.

Normally, the VLSiCE-96P system traces program activity while the user program executes. With a trace specification, tracing can be specified to occur only when specific conditions are met during execution. The trace is collected in a buffer that can collect data on up to 4K entries of information during emulation.

The trace buffer can be examined during halt mode or if non-stop emulation is desired, the trace can be examined while emulation continues. If this second option is selected, trace collection stops briefly while the trace buffer is uploaded to the host.

Arming and Triggering

The VLSiCE-96P command language allows specification of complex events with up to 8 states, each with several conditions. For example, a specification can be made that causes a break to occur when a variable is written only within a certain procedure. The execution of the procedure is the arm condition; the variable modification is the break condition. The arm condition is an optional part of a break/trace sequence in the VLSiCE emulator. A set of arm conditions can be used to ensure that a system break is not possible until all required qualifying conditions are satisfied.

Procedures

Debugging procedures (PROCS) are a user-named group of VLSiCE commands that are executed sequentially. Procs can simulate missing hardware or software, collect debug information, and make troubleshooting decisions. They can be copied to text files on disk, then included from the file into the command sequence in later test sessions.

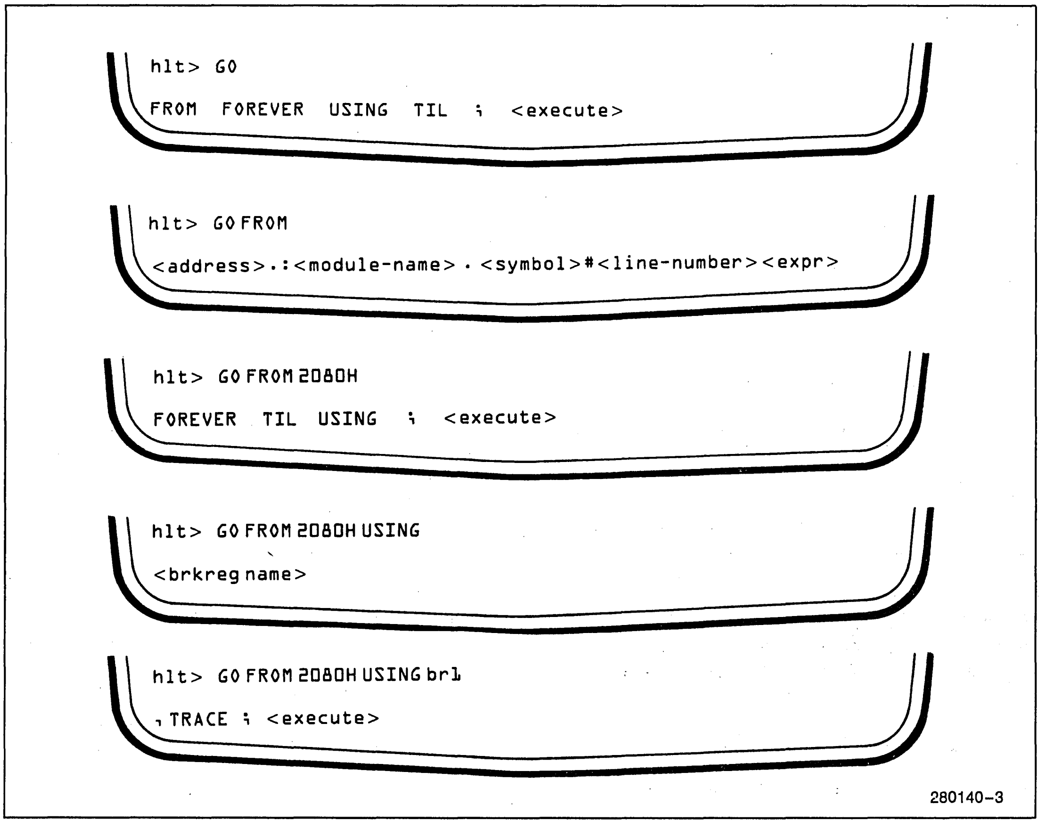
Procedures can also serve as programmable diagnostics, implementing new emulator commands for special purposes or to increase generality.

Dynamic Tracing

The trace buffer can be dynamically accessed during emulation. Any form of the PRINT command can be entered and the specified portion of the trace buffer is displayed. This allows real-time display of processor activity. However, displaying the trace buffer during emulation stops collection of trace and some trace information could be lost.

On-Line Syntax Guide

A special syntax guide called the Integrated Command directory (ICD), at the bottom of the display screen, aids in creating syntactically correct command lines. Figure 3 shows an example of the ICD for the GO command.



```
hlt> GO
FROM  FOREVER  USING  TIL  ;  <execute>
```

```
hlt> GO FROM
<address>.:<module-name> . <symbol>#<line-number><expr>
```

```
hlt> GO FROM 2080H
FOREVER  TIL  USING  ;  <execute>
```

```
hlt> GO FROM 2080H USING
<brkreg name>
```

```
hlt> GO FROM 2080H USING br1
, TRACE ; <execute>
```

280140-3

Figure 3. The Integrated Command Directory for the GO Command

HELP

This feature provides assistance with the emulator commands through the host system terminal. HELP is available for most of the commands. Figure 4 shows one of the commands HELP can be obtained for.

	Min.	**Max.**
Clock Frequency	6 MHz	10 MHz
V_{CC}	4.75V	5.25V
I_{CC}(1)		400 mA

NOTE:
1. All outputs disconnected.

DESIGN CONSIDERATIONS

There are design considerations that the user should be aware of before designing with the VLSiCE-96P emulator.

Electrical Considerations

The user pin timings and loadings are identical to the 8096 component except as noted below. Also, the RESET and CLKOUT pins have an additional loading of 1 μA and 10 pF. The Non-Maskable Interrupt (NMI) is not supported.

Mechanical Considerations

The user plug is at the end of a three foot flexible cable. Adequate spacing must be provided on the target system to allow the processor module to be inserted into the target system.

The height of the processor module and target adaptor may pose a problem for multiple board system prototypes that need to be debugged and tested. Be sure that the space between the boards is greater than 1½" to allow for the processor module and target adaptor.

```
hlt> HELP ASM
The ASM command displays or modifies memory as 8096 assembler mnemonics.
The syntax is:

 ASM<partition>     ::=<partition> [='<asm96-inst>'    [,'<asm-96inst>'*]]
                    <address> <cr>

where:

<partition>specifies the area of memory to be displayed and modified.

<asm96-inst>specifies the 8096 assembly instruction to be assembled.

<address> is any valid 8096 address.

<cr> indicates a carriage return.

The "ASM <address>=" syntax puts the user in line-mode, displaying the
current address at which the instruction will be placed and not requiring the
quotes around the instructions.

Please see the VLSiCE-96P Reference Manual for more detailed information.
```

Figure 4. HELP Screen

Figure 5 shows the dimensions for the processor module and target adaptor. In the figure, please note the location of pin 1 on each target adaptor.

Limitations and Restrictions

- The non-maskable interrupt (NMI) is not supported.
- High Speed Input/Output (HSIO) is emulated, but can not be read or written to during interrogation mode.
- If a break occurs immediately before the JVT instruction, the VT flag is cleared. There is no way to then tell if the flag was set before re-entering emulation.

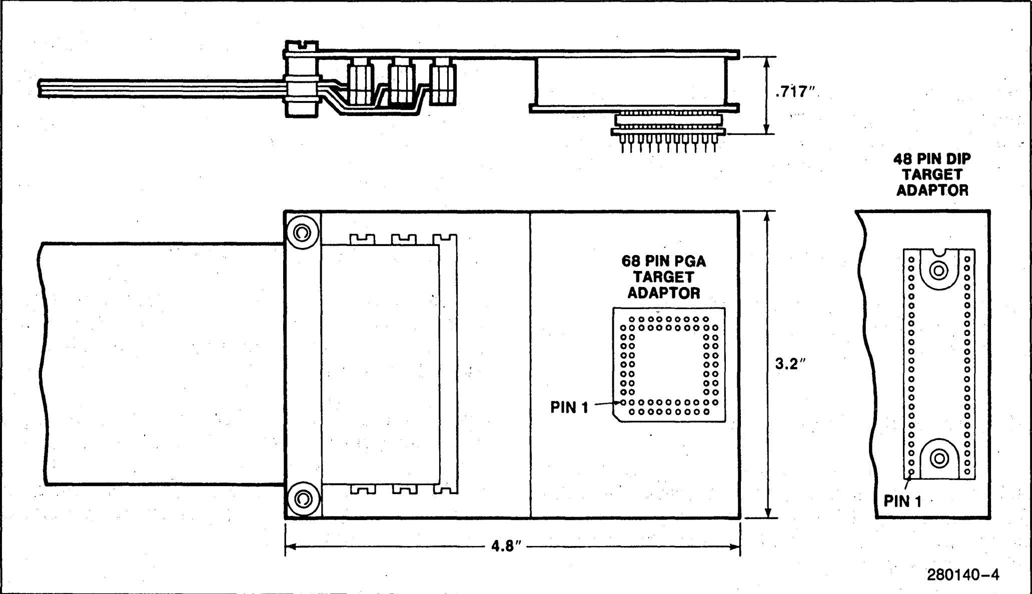


Figure 5. Dimensions for the Processor Module and Target Adaptor

- If a read access of SP_STAT or IOS1 occurs at the same time that an interrupt sets a bit in the accessed register, there is a possibility that the new interrupt flag will be lost, because read accesses to the SP_STAT or IOS1 registers clear these registers. A software work-around for this problem is provided.
- Bit 7 of Port 0 is inverted when the digital value of the port is read.
- The counters for the pulse width modulator (PWM) and hardware Timer 1 can be out of sync if either are disabled during interrogation. They can be re-synced with a user reset of the system.
- The READY pin is not supported. It should be tied high (+5V).
- The system presently operates up to 10 MHz.
- Ports 3 and 4 cannot be read by code during emulation or by the host during interrogation when they are used as ports (EA/=1).

The VLSiCE-96P system has some limitations that are inherent in the 809X-90 core of the emulation processor. These limitations are:

- The displacement portion of an indexed, three word multiply may not be in the range of 200H through 17FFH inclusive.
- The EXT instruction never sets the N flag, and always sets the Z flag. The EXTB instruction works correctly.
- The zero flag is either set or cleared by the Add or Subtract with carry instructions as appropriate.
- Trying to read, modify, or write the interrupt pending register may cause interrupts to be missed during execution of the instruction.
- The V and VT flags may indicate an overflow after a signed divide when no overflow has occurred.
- If an event on an HSI pin set to look for every eighth transition occurs less than 16 state times after an event on any other pin, then the divide by 8 event is recorded twice in the HSI FIFO. The time tag of the duplicate FIFO entry is equal to that of the initial entry plus one.
- An event occurring within 16 state times of a prior event on the same HSI line may not be recorded. Additionally, an event occurring within 16 state times of a prior event on another HSI line may be recorded with a time tag one count earlier than expected. Events are defined as the condition the line is set to trigger on.
- Software timer interrupts cannot be generated by the HSO commands that reset Timer 2 or start an A to D conversion.
- The serial port is not tested in mode 0. The receive function in this mode does not work correctly.
- Loading the baud rate register with 8000H (maximum baud rate, internal clock) may cause an 11 millisecond delay (at Fosc = 12 MHz) before the port is properly initialized. After initialization the port works properly.
- To be used as outputs, ports 3 and 4 each must be addressed as words but written to as bytes. To write to Port 3 use 'ST temp, 1ffeh', where the low byte of 'temp' contains the data for the port. To write to Port 4, use the DCB operator to generate the opcode sequence 0C3H, 001H, 0FFH, 01FH, (temp), where the high byte of temp contains the data for the port. Ports 3 and 4 do not work as inputs.
- The watchdog timer does not run after a chip reset until a 01EH followed by a 0E1H is written to the watchdog timer register. When this is done, the watchdog timer functions as described in the 8096 Users Manual until the next chip reset. This feature permits disabling of the watchdog by not writing to it.
- External interrupts on P0.7 are sampled every state time instead of every eight state times.
- The baud rate generated in the external clock can be computed using this formula:

Baud Rate = Input Frequency/(16*B)

This formula does not work in mode 0 and assumes T2CLK is the source of the input frequency.

- When more than one HSO event occurs with interrupts occurring at the same time, multiple HSO interrupts may occur. This is because HSO interrupts are internal events and are not synchronized to Timer 1.
- Locations 2012H through 207FH in external memory must be filled with the hex value 0FFFFH to ensure compatibility with future parts. The internal locations in this range are still reserved for the factory test code.
- Neither the source nor the destination addresses of the Multiply or Divide instructions can be a writable special function register.
- The special function registers may not be used as base or index registers for indexed or indirect instructions.
- Several of the special function registers can only be accessed as words, while others only as bytes. These restrictions are listed in the 8096 Users Manual.

SPECIFICATIONS

Host Requirements

Disk drives— Dual floppy or 1 hard disk and 1 floppy drive required (10M-byte Winchester optional).

An IBM PC XT or PC AT with 512K RAM and hard disk. Intel recommends PC-DOS 3.0 or later. Earlier versions of PC-DOS may be acceptable.

Other peripherals as desired.

VLSiCE-96P Software Package

VLSiCE-96P emulator software
VLSiCE-96P confidence tests
VLSiCE-96P tutorial software

System Performance

Mappable zero wait-state memory (zero wait-states up to 12 MHz)	Min. 0K-bytes Max. 64K-bytes	Mappable to user or ICE memory in 1K blocks on 1K boundaries.
Trace buffer	4K × 48 bits	
Virtual symbol table—	A maximum or 61K-bytes of host memory space is available for the virtual symbol table (VST). The rest of the VST resides on disk and is paged in and out as needed.	

Physical Characteristics

Controller Pod

Width: 8¼" (21 cm)
Height: 1½" (3.8 cm)
Depth: 13½" (34.3 cm)
Weight: 4 lbs (1.85 kg)

Power Supply

Width: 7⅝" (18.1 cm)
Height: 4" (10.06 cm)
Depth: 11" (27.97 cm)
Weight: 15 lbs (6.1 kg)

User Cable

3′ (.944m)

Serial Cable

12′ (3.6m)

Electrical Characteristics

Power Supply

100–120V or 220–240V (selectable)
50–60 Hz
2 amps (AC max)@ 120V
1 amp (AC max)@ 240V

Environmental Characteristics

Operating temperature:	0°C to 40°C (32°C to 104°F)
Operating humidity:	Maximum of 85% relative humidity, non-condensing

DOCUMENTATION

VLSiCE-96P In-Circuit Emulator User's Guide, order number 165814.

VLSiCE-96P In-Circuit Emulator Installation Supplement for Intel Hosts, order number 166477.

VLSiCE-96P In-Circuit Emulator Installation Supplement for IBM Hosts, order number 166478.

ORDERING INFORMATION

Base Hardware and Software

Order Code	Description
V096S96A	VLSiCE-96P emulation base and software on 8″ single density media for hosting on an Intel Series III. [Requires software license].
V096S96B	VLSiCE-96P emulation base and software on 8″ double density media for hosting on an Intel Series III. [Requires software license].
V096S96C	VLSiCE-96P emulation base and software on 5¼″ media for hosting on an Intel Series IV. [Requires software license].
V096S96D	VLSiCE-96P emulation base and software on 5¼″ media for hosting on an IBM PC-AT under PC-DOS 3.0. [Requires software license].

Serial Cables (must be ordered for complete system)

Order Code	Description
SCOM1	Serial communications cable is a 25-pin male to 9-pin male for hosting VLSiCE-96P on Intel Series III/IV.
SCOM2	Serial communications cable is a 25-pin female to 9-pin male for hosting VLSiCE-96P on IBM PC-XT.
SCOM3	Serial communications cable is a 9-pin female to 9-pin male for hosting VLSiCE-96P on IBM PC-AT.
TA096E	68 pin PLCC adaptor board.

Software Only

Order Code	Description
S096AP	Software for host, probe, diagnostic and tutorial on 8″ single sided media for use with a Series III. [Requires software license].
S096BP	Software for host, probe, diagnostic and tutorial on 8″ double sided media for use with a Series III. [Requires software license].
S096CP	Software for host, probe, diagnostic and tutorial on 5¼″ media for use with a Series IV. [Requires software license].
S096DP	Software for host, probe, diagnostic and tutorial on 5¼″ media for use with a IBM PC AT or PC XT (requires PC DOS 3.0 or later). [Requires software license].

Other Useful Intel MCS®-96 Debug and Development Support Products

Order Code	Description
I86ASM96	Consists of the ASM 96 macro assembler that translates symbolic assembly language mnemonics into relocatable object code, and the RL96 linker and relocator program that links modules generated by ASM 96 and PL/M 96 and locates the linked object modules to absolute memory locations. System requirements and Intellec® System running iNDX.
I86PLM96	Consists of the PL/M 96 compiler that provides high level programming language support, the LIB 96 utility that creates and maintains libraries of software object modules, the FPAL96 floating point arithmetic library, and the RL96 linker and relocator program that links modules generated by ASM 96 and PL/M 96 and locates the linked object modules to absolute memory locations. System requirements and Intellec® System running iNDX.
D86ASM96NL	ASM/R&L 96 for PC-DOS. It contains a macro assembler, a linker/locator utility, a floating point utility and a librarian. System requirements are an IBM PC AT or PC XT with 512 K of RAM and PC-DOS 3.0 or greater.

Order Number	Description
D86PLM96NL	PL/M 96 and R&L for PC-DOS. It contains a compiler, a linker/locator utility, a floating point utility and a librarian. System requirements are an IBM PC AT or PC XT with 512K of RAM and PC-DOS 3.0 or greater.
D86C96NL	C96 and R&L for PC-DOS. Contains a compiler linker/locator utility, and all standard C libraries, including STDIO. System requirements are an IBM PC AT or PC XT with 512K of RAM and PC-DOS 3.0 or greater.
SBE96SKIT	iSBE-96 single board emulator for use with the Series III/IV development systems. The kit contains: iSBE-96 Single Board Emulator; MCS®-96 software support package for the Series III/IV development systems. iSBE-96 Series III/IV upgrade kit (cables and software needed to run on Intel Hosts).
SBE96DKIT	iSBE-96 single board emulator for use with the IBM PC AT and PC XT computer systems. The kit contains: iSBE-96 single board emulator; MCS®-96 software support package for PC-DOS. iSBE-96 DOS upgrade kit (cables and software needed to run on the IBM PC AT or PC XT).

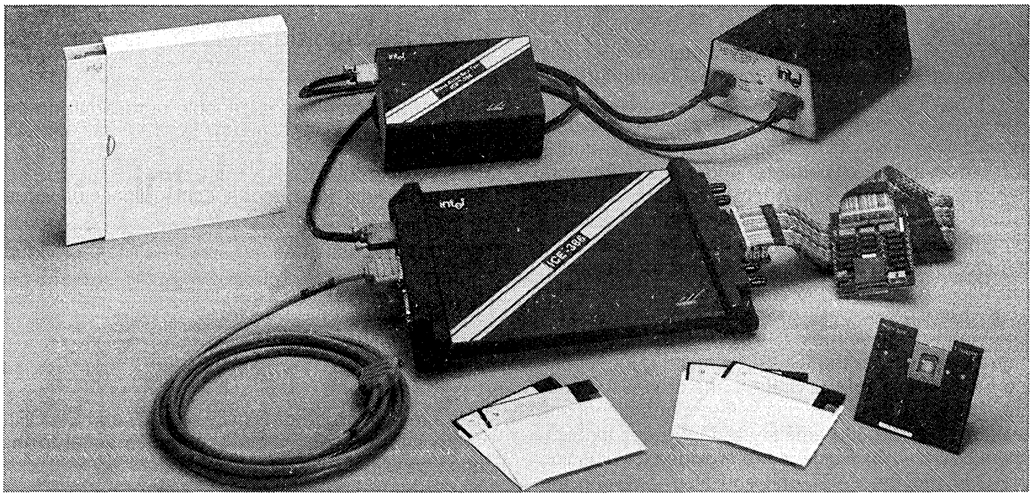
Running the iSBE-96 emulator on the Series II and iPDS system requires software from:

U.S. Software Corporation
5470 N.W. Innisbrook
Portland, OR 97229
Phone: 503-645-5043
International Telex: 4993875

ICE™-386
In-Circuit Emulator for the 80386

- Provides Real-time Emulation of the 80386 at Speeds up to 16 MHz
- Maps User Program Memory into a Maximum of 128 KB of Memory in 4 KB Increments
- Allows Zero Wait-state Operation From User System Memory
- Provides Full Debug Support of 8086 Absolute Intel OMFs, and the 80286 and 80386 Bootloadable Intel OMFs
- Hosted on the Intel 286/310 System Running XENIX* or on the IBM PC AT Running DOS
- Allows User to Examine and Modify Memory, I/O, and the 80386 Registers
- Allows User to Examine and Modify Descriptor Tables, Page Tables, the Interrupt Table, and the Task State Segment
- Allows User to Set Breakpoints on Task Switching, External Inputs, Trace Buffer Full, and Instruction Execution
- Provides Real-time Execution Trace With Time Tags That Can Be Used to Analyze Execution Time of User Code
- Supports Coprocessor Debugging
- Stores over 2000 Frames of Program Execution History
- Allows User to Assemble and Disassemble Program Code in 80386 Instruction Mnemonics
- Provides Human Interface with Command Line Syntax, Guidance, Command Line Editing, Control Constructs, Debug Procedures, Debug Variables, and Shell Escape to the Host Operating System
- Uses the Four On-chip Breakpoints to Recognize Instruction Execution Addresses or Data Access Addresses Which Can Be Used to Trigger Break, Trace, or Change the State of External Sync Lines

The ICE-386 In-Circuit Emulator provides sophisticated hardware and software debugging capabilities for 80386-based designs. These capabilities include emulation of the 80386 CPU and the ability to examine and modify registers in optional numeric coprocessors.



280316-1

*XENIX is a registered trademark of Microsoft Corporation.
‡UNIX is a registered trademark of AT&T Bell Laboratories.

November 1986
Order Number: 280316-002

PHYSICAL DESCRIPTION

The ICE-386 emulator consists of the following (see Table 1):

- Power supply with three power cables (an AC cable and two DC cables)
- Control unit (CU) with an RS-232C communication cable
- User cable assembly with Processor Module (PM)
- Stand-alone/self-test (SAST) unit
- Optional isolation board (OIB)
- A signal access board (SAB)
- SAB removal tool
- Software (includes the emulator software and diagnostic software)

The power supply powers the control unit and the stand-alone self-test unit (SAST).

The control unit (CU) controls the processor module and communicates with the host. The control unit has 128 KB of emulation memory, over 2000 frames of trace storage, and a control processor. The following connectors are on the control unit:

- Power supply connector
- GPIB connector
- RS-232C serial connector
- User cable connector
- Four BNC connectors, two for SYNCIN and two for SYNCOUT signals (used as qualifiers for triggering external events with the GO command).

The RS-232C cable or the (optional) GPIB cable connects the control unit to the host computer.

The user cable assembly comprises the processor module (the PM, which includes the 80386 emulation processor and IC buffers) and a cable that connects to the control unit. A target adaptor, mounted to the bottom of the PM, provides pin compatibility with the target 386 socket. The PM also has a socket for connecting the OIB or SAST unit to the emulator.

The stand-alone self-test (SAST) unit permits stand-alone operation of the ICE-386 emulator by supplying power and a 16MHz CLK2 to the processor module. The SAST unit provides self-test and diagnostic tests for the ICE-386 emulator and the optional isolation board (OIB). Sockets are provided for plugging the OIB into the SAST unit and also for using the ICE-386 emulator with math coprocessors.

The optional isolation board (OIB) is a circuit board that is installed between the processor module and the target system to protect the 80386 emulation processor from an untested target system. This ensures proper operation of the ICE-386 emulator even if the user bus fails. When it is determined that the target system will not electrically damage the 80386 emulation processor, the OIB can be removed and the emulation processor can operate at full speed. Use of the OIB limits the operating speed of the emulator to a maximum frequency of 8 MHz. See Table 3 for complete details on OIB timing.

The signal access board (SAB) provides access to all the 80386 signal pins for attaching external instruments, e.g., a logic analyzer. It can be installed between the processor module and the OIB board, or between the processor module and the target system.

The SAB removal tool is used to separate the signal access board from the processor module.

FUNCTIONAL DESCRIPTION

16 MHz Emulation

The ICE-386 emulates the 80386 at speeds up to 16 MHz, thus providing early detection of subtle timing problems that may arise at full speed. Intel's bond-out technology stresses the tightest possible conformance between timing parameters of the emulator and the target processor.

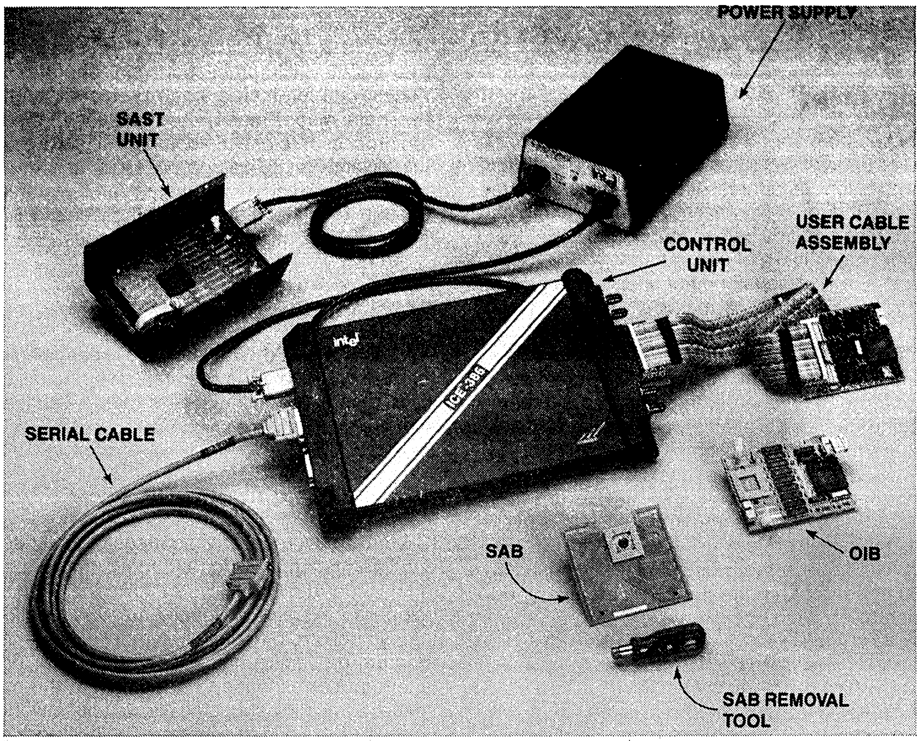
Event Recognition and Emulation Control

The ICE-386 emulator is capable of recognizing the following conditions:

- An instruction boundary
- A data write to a user-specified linear or virtual address
- A data access at a user-specified linear or virtual address
- Trace buffer full
- Task switch
- External input

Data breaks can be of BYTE, WORD, and DWORD length and are aligned on boundaries equal to the range size.

Table 1. ICE™-386 System Overview



280316-2

Name	Description
Control Unit	Contains an 80188 control processor, serial and GPIB communications, 128 KB mappable memory, and a trace buffer.
User Cable Assembly	Includes the user cable and the processor module, which contains the 80386 emulation processor and the OIB/SAST connector.
SAST Unit	Allows stand-alone operation and self-diagnosis of the ICE-386 emulator.
OIB	Protects the processor module from user-target-board bus failures during early prototype development.
SAB	Provides access to the labeled 80386 pins for use with external instruments.
SAB Removal Tool	Allows easy removal of the SAB from the emulation processor.

Based on the preceding events, the ICE-386 emulator can take the following actions:

- Halt emulation
- Sequence through events; maintain a count of such occurrences
- Assert SYNCOUT lines
- Turn trace on/off

In addition, software breakpoints, which allow breaks on instruction execution only, are useful in applications requiring extensive emulation breaking. These breakpoints have the following characteristics:

- Software breakpoints are placed only on RAM-based program code.
- Software breakpoints are valid in any addressing mode; however, if they are placed on a virtual/linear address, they will be translated to a physical address.
- Software breakpoints cannot be altered by the GO command; they can only be removed.

The ICE-386 emulator requires exclusive use of the 80386's on-chip debug registers and INT1 under normal usage.

Memory Mapping

Memory can be mapped either to the user hardware (USER) or to the ICE-386 emulator (ICE). The 80386 microprocessor can address four gigabytes of memory space. Partitions anywhere within this four gigabytes processor space can be mapped to ICE. Thus, you can replace blocks of user address space to ICE and begin software development prior to completion of the target hardware. A maximum of 128 KB of mapped memory with an average of six wait states is mappable in 4 KB increments on 4 KB boundaries. Program code can be executed at zero wait-state from the target memory. When using mapped memory, paging cannot be enabled because both functions share the same resources on the 80386.

Mapping memory to ICE enables software development to begin before a user memory system is available. If the target system uses EPROMs, ICE-386 memory can replace the EPROM memory space so that you need not program the EPROMs during development.

Program Tracing

Over two thousand frames of program execution history can be stored in the trace buffer. Each trace frame holds a linear address of the branch address (or old TSS and new TSS for task switch), the byte count between branches, and a time-tag. This information is used to reconstruct a history of program execution. The ICE-386 emulator runs at about 93% of its full speed (when trace is enabled) assuming that a program discontinuity occurs every seven instructions. Turning the trace option off enables true full-speed emulation of the 80386 microprocessor. Trace data can also be displayed during emulation. However, enabling this feature may degrade emulator performance by causing the 80386 to enter a ready hang while the ICE-386 control processor is accessing the trace buffer.

Processor/Memory Examination and Modification

80386 registers can be accessed mnemonically (e.g., EAX) with the ICE-386 emulator software. Data can be displayed or modified in one of four bases: hexadecimal, decimal, octal, or binary, and in ASCII format. Program code can be disassembled and displayed as 80386 assembly instruction mnemonics, or it can be disassembled and displayed in 8086 or 80286 mode.

The 80386 microprocessor can operate in four modes: real, protected, page protected, and virtual 86. These modes modify the operation of the emulation processor to provide compatibility with the 8086, 80186, and 80286 microprocessors and to provide support for a variety of system architectures. The ICE-386 emulator can be used to debug in all of these microprocessor operating modes.

Program Stepping

With the ICE-386 emulator, you can single step through program code by referencing machine-level instructions, line numbers, or high-level language statements.

Symbolic Debugging

ICE-386 software takes advantage of the special debug information provided by Intel compilers, providing superior debug ability. It allows the software developer to examine or modify memory locations using symbolic references. A symbolic reference is a program procedure name, line number, or program label that corresponds to a location in the program space. Symbolic debugging allows you to work in the context of the original program, helping to meet the most critical schedules.

The ICE-386 emulator maintains a virtual table for program symbols making it possible for the table to exist without fitting entirely into the host RAM memory.

ICE™-386 Emulator Operating Modes

The ICE-386 emulator software has two operating modes, interrogation and emulation. Interrogation mode allows you to enter any ICE-386 command. Emulation mode allows execution of the user program code and execution of commands not requiring interaction with the emulation processor.

Debug Procedures

Debug procedures (PROCs) are named, user-defined groups of ICE-386 emulator commands. They can be stored on disk and recalled in later debugging sessions, thus saving you from having to re-enter commands.

One advantage of PROCs is that they allow you to automate the software testing process. The PROC may repeatedly generate test values, execute the user program with varied input values, and record the results. PROCs aid in the development of comprehensive batch tests.

ICE™-386 Human Interface Features

The ICE-386 emulator software includes features to help enter commands, set up the debug environment, and display command options. Also, there are features similar to those found in the UNIX‡/XENIX operating system environment. These features include input/output redirection, command piping, and escaping from the shell. The ICE-386 emulation software has a set of mathematical operators and control constructs similar to those found in Intel's I²ICE™ emulator and the C programming language.

Coprocessor Support

ICE-386 emulator commands provide access to the coprocessor's stack, status registers, and flags. In addition, the disassembly command extends to the math coprocessor's instructions and data types.

Language Support

The ICE-386 supports all 8086 absolute and 80286/80386 bootloadable Intel OMFs. These include the following Intel languages:

ASM86	ASM286	ASM386
PL/M-86	PL/M-286	PL/M-386
C-86	C-286	C-386
Pascal-86	Pascal-286	
FORTRAN-86	FORTRAN-286	

ICE-386™ EMULATOR COMMAND FUNCTIONS

The ICE-386 emulator command language can be divided into the following functional categories.

- Emulation commands — The GO, LSTEP, PSTEP, and ISTEP commands instruct the emulator to begin emulation. The GO command also has many options that allow recognition of a complex set of conditions.
- Utility commands — These are general purpose commands for use in a debugging environment. For example, the PRINT command displays selected frames of trace data. A command line editor and history buffer are also provided for ease of command entry.
- Environment commands — These commands set up the debug environment. For example, the MAP command sets up the memory map. The DEFINE command is used to create PROCs and definitions.
- File handling commands — These commands are used to access disk files. Debug object definitions can be saved to a disk file for use in later debug sessions. Debug sessions can be recorded to a disk file or a line printer for later analysis.
- Register access commands — These commands enable you to access the 80386 processor registers. Registers may be displayed or modified by name (e.g., EAX, EBX, etc.) or may be displayed in groups (REGS, CREGS, and SREGS).
- Descriptor access commands — These commands enable you to display descriptor tables and to symbolically display or modify individual descriptor components (e.g., GDT[10].limit, DT[CS].base).
- Memory and I/O access commands — These commands provide access to user memory space. Input and output are interpreted according to a set of data types and the requested numeric base.
- Stack frame access command — CALLSTACK allows you to display the current chain of procedure calls in the user program being executed.

SPECIFICATIONS

Host Specifications

The following lists the minimum host requirements for the Intel 286/310 and the IBM PC AT.

- Intel System 286/310 — 2 MB of RAM, 40 MB fixed disk, at least one floppy disk drive, and the XENIX 286 operating system (version 3.0 with Update 3).
- IBM PC AT — 2 MB of RAM (Lotus*, Intel, and Microsoft† extended memory specifications) (Intel's ABOVE Board with 1.5 MB is required), 20 MB fixed disk, at least one floppy disk drive, a serial interface, and the DOS operating system (version 3.1).

For enhanced upload/download performance, an optional IEEE-488 (GPIB) adaptor can be purchased for the IBM PC AT. Please contact your local Intel Salesperson for details.

Mechanical Specifications

PROCESSOR MODULE/OPTIONAL ISOLATION BOARD DIMENSIONS

The height of the processor module (with OIB or SAB installed) requires 1¼ inches (3.18 cm) of space between boards to connect the processor module to the OIB, or to connect the processor module to the SAB. If the processor module is connected to both the OIB and the SAB, 1¼ inches (3.18 cm) are required on both sides, thus increasing the space requirement to 2½ inches (6.35 cm). Figure 1 shows the dimensions of the procesor module, the OIB and the SAB in the three possible configurations.

Table 2 lists the physical characteristics of the ICE-386 emulator components.

Electrical Specifications

SYNC LINE SPECIFICATIONS

The SYNCIN lines must be valid for at least one instruction cycle because they are only sampled on instruction boundaries. The SYNCOUT lines are driven by TTL open collector outputs that have 4.75K-ohm pull-up resistors. The SYNCIN lines are standard TTL inputs.

AC SPECIFICATIONS

Table 3 lists the ICE-386 emulator's AC specifications with the OIB installed. In Table 3, the numbers shown in the column titled "Maximum" are derived from using the timing symbols (t1, t2a, t2b, etc.) in the AC Specification Tables in the *80386 High Performance 32-Bit Microprocessor With Integrated Memory Management* data sheet and adding the additional time required to operate when the OIB is installed. The asterisks following the timing symbols in the column titled "Symbol" indicate that these timing symbols relate to the timing symbols of the 80386 microprocessor operating at 16 MHz. Active low is indicated by the overscore (e.g., $\overline{\text{LOCK}}$).

The AC specifications for the ICE-386 emulator are the same as those for the 80386 microprocessor, with the 01B removed, except for t25 and t26, as shown in Table 4.

DC SPECIFICATIONS

Table 5 lists the DC specifications of the ICE-386 emulator at the processor module with the OIB installed.

Table 6 lists the DC specifications of the ICE-386 user probe with the OIB board removed.

The ICE-386 emulator's buffer circuitry adds additional DC loadings to the 80386 pins listed in Table 7.

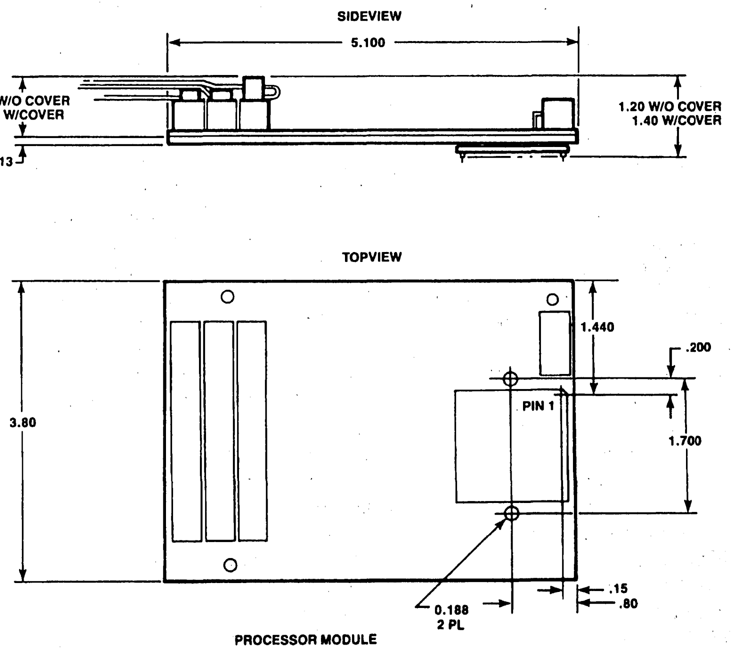
Power Supply

100–120V or 220–240V (selectable)
50–60 Hz
2 amps (AC Max) @ 120V
1 amp (AC Max) @ 240V

Table 2. ICE™-386 Physical Characteristics

Unit	Width		Height		Length		Weight	
	inches	cm	inches	cm	inches	cm	lbs	kg
Control unit	10.5	26.7	1.5	3.8	16.0	40.0	6.0	2.72
Processor module†	3.8	9.6	1.3	3.3	5.1	13.0		
SAST*	6.0	15.2	2.0	5.1	8.0	20.3	3.5	1.59
OIB	3.8	9.6	.9	2.3	5.1	13.0		
Power supply	2.8	7.1	4.15	10.7	11.0	27.9	4.7	2.14
User cable					22.0	55.9		
Serial cable					12.0′	3.66m		
SAB	4.6	11.7	.8	2.0	4.1	10.7		

†Measurement includes target adapter
*Measurement includes user cable



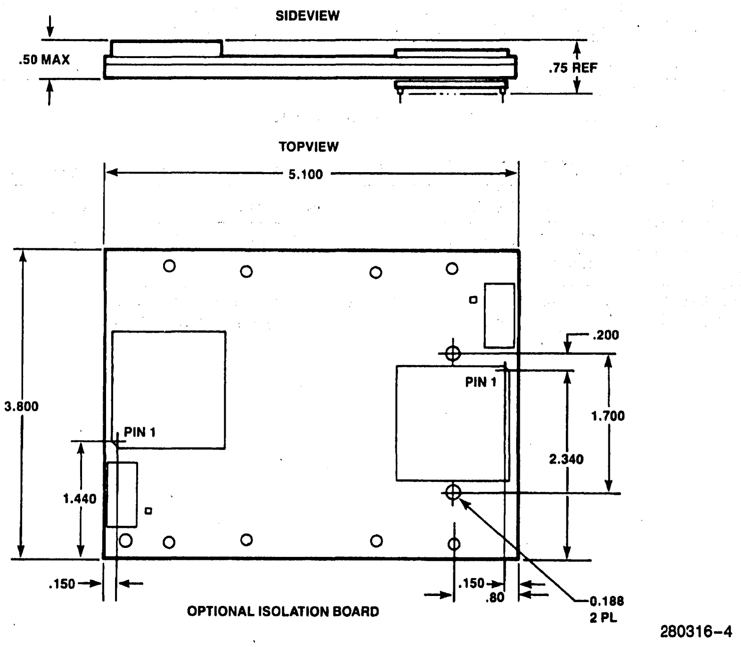


Figure 1. Board Dimensions

Table 3. AC Specifications With the Optional Isolation Board Installed

Symbol	Parameter	Minimum	Maximum	Notes
t1*	CLK2 period	62.5 ns	t1 Max	
t2a*	CLK2 high time	t2a Min + 2 ns		@2V
t3b*	CLK2 low time	t3b Min + 2 ns		@0.8V
t6*	A2–A31 valid delay	t6 Min + 2 ns	t6 Max + 16.1 ns	CL = 120 pf
t7*	A2–A31 float delay	t14 Min + 6 ns	t14 Max + 14.8 ns	
t8*	$\overline{\text{BE0}}$–$\overline{\text{BE3}}$, $\overline{\text{LOCK}}$ valid delay	t8 Min + 2 ns	t8 Max + 14.8	CL = 75 pf
t9*	$\overline{\text{BE0}}$–$\overline{\text{BE3}}$, $\overline{\text{LOCK}}$ float delay	t14 Min + 6 ns	t14 Max + 27	
t10*	$\overline{\text{W/R}}$, $\overline{\text{M/IO}}$, $\overline{\text{D/C}}$, $\overline{\text{ADS}}$ valid delay	t10 Min + 2 ns	t10 Max + 14.8	CL = 75 pf
t11*	$\overline{\text{W/R}}$, $\overline{\text{M/IO}}$, $\overline{\text{D/C}}$, $\overline{\text{ADS}}$ float delay	t14 Min + 6 ns	t14 Max + 27	
t12*	D0-D31 write data valid delay	t12 Min + 3 ns	t12 Max + 12.1 46 ns	CL = 120 pf Note 1
t13*	D0-D31 write data float delay	10 ns	41 ns	
t14*	HLDA valid delay	t14 Min + 1 ns	t14 Max + 6 ns	
t16*	$\overline{\text{NA}}$ hold time	t16 Min + 6 ns		
t18*	$\overline{\text{BS16}}$ hold time	t18 Min + 6 ns		
t20*	$\overline{\text{READY}}$ hold time	t20 Min + 6 ns		
t21*	D0–D31 read setup time	t21 Min + 9 ns		
t22*	D0–D31 read hold time	t22 Min + 3 ns		
t23*	HOLD setup time	t23 Min + .5 ns		
t24*	HOLD hold time	t24 Min + .5 ns		
t25*	RESET setup time	t25 Min + .5 ns		Note 2
t26*	RESET hold time	t26 Min + .5 ns		Note 2
t27*	NMI, INTR setup time	t27 Min + .5 ns		
t28*	NMI, INTR hold time	t28 Min + .5 ns		
t29*	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ setup time	t29 Min + .5 ns		
t30*	PEREQ, $\overline{\text{ERROR}}$, $\overline{\text{BUSY}}$ hold time	t30 Min + .5 ns		

Note 1: t12* Max is the larger of the two entries.
Note 2: To calculate t25* or t26*, use the value of t25 or t26 in Table 4.

Table 4. ICE™-386 Emulator AC Specifications With the OIB Removed

Symbol	Parameter	Minimum	Maximum	Notes
t25	RESET setup time	12 ns	—	1
t26	RESET hold time	3 ns		1

Note 1: These setup and hold specifications must be satisfied to ensure proper syncronization of the ICE-386 emulator to the target system. The RESET signal is delayed two or four CLK2 cycles before arriving at the RESET input of the 80386.

Table 5. ICE™-386 Emulator DC Specifications With OIB Installed

Symbol	Parameter	Minimum	Maximum	Notes
OIB-I_{CC}	Supply current		PM-I_{CC} + 500 mA	1
V_{ol}	Output low voltage. I_{ol} ▪ 48 ma, A2-A31, D0-D31, $\overline{BE0}$-$\overline{BE3}$, $\overline{ADS}$, $\overline{W/R}$, $\overline{D/C}$, $\overline{M/IO}$, $\overline{LOCK}$, HLDA		0.5V	
V_{oh}	Output high voltage. I_{oh} = 3ma, A2-A31, D0-D31, $\overline{BE0}$-$\overline{BE3}$, $\overline{ADS}$, $\overline{W/R}$, $\overline{D/C}$, $\overline{M/IO}$, $\overline{LOCK}$, HLDA	2.4V		
I_{il}	Input low current. CLK2, RESET, HOLD $\overline{READY}$		.5 mA .26 mA	
I_{ih}	Input high current. CLK2, RESET, HOLD $\overline{READY}$		20 μa 40 μa	
I_{lO}	Output leakage current. A2-A31, D0-D31, $\overline{BE0}$-$\overline{BE3}$, $\overline{ADS}$, $\overline{W/R}$, $\overline{D/C}$, $\overline{M/IO}$, $\overline{LOCK}$		± 20 μa	

Note 1: OIB-I_{CC} is the supply current of the ICE-386 user probe with the OIB installed. PM-I_{CC} is the supply current of the user probe with the OIB removed.

Table 6. ICE™-386 Emulator DC Specifications Without OIB Installed

Symbol	Parameter	Maximum	Notes
PM-I_{CC}	Supply current. CLK2 = 16 MHz: with 80386-12	80386-I_{CC} + 800 mA	
C_{in}*	Input capacitance	C_{in} + 15 pf	1, 2
C_{out}*	Output or I/O capacitance	C_{out} + 15 pf	1, 3
C_{clk}*	CLK2 Capacitance	C_{clk} + 15 pf	1

Note 1: Symbols followed by an asterisk are ICE-386 user-probe specifications and are given in terms of the corresponding 80386 specifications plus a constant.
Note 2: Specification applies to the READY and RESET signals.
Note 3: Specification applies to t he A2-A31, D0-D31, $\overline{BE0}$–$\overline{BE3}$, $\overline{W/R}$, $\overline{M/IO}$, $\overline{ADS}$, and HLDA signals.

Table 7. Additional DC Loading

Signal	I_{ih} Maximum	I_{il}Maximum
A2-A31, D0-D31, $\overline{BE2}$, $\overline{BE3}$	0.02 mA	0.1 mA
$\overline{ADS}$, $\overline{BE0}$, $\overline{BE1}$, HLDA	0.02 mA	1 mA
$\overline{M/IO}$	0.02 mA	0.5 mA
$\overline{W/R}$	0.04 mA	1 mA
$\overline{READY}$	0.02 mA	0.02 mA
CLK2	0.05 mA	2 mA

Note: For more information on these signals, refer to the 80386 data sheet.

Environmental Specifications

Operating temperature — 10°-40°C (50°-104°F)

Operating humidity — maximum of 85% relative humidity, non-condensing

DOCUMENTATION

ICE™-386 In-Circuit Emulation User's Guide, Order Number 166182

80386 High Performance 32-Bit Microprocessor with Integrated Memory Management, data sheet, Order Number 231630

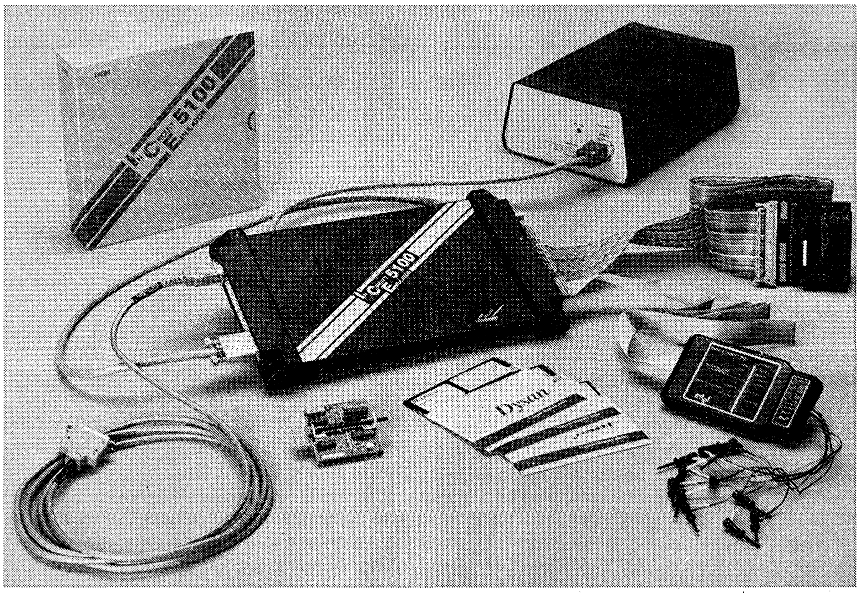
ORDERING INFORMATION

Order Code	Description
ICE386XP ICE386X	The complete ICE-386 emulator system including control unit, processor module, power supply, SAST, OIB, SAB, a serial communication cable (SCOM5), and software (S386XP/S386X). (Requires software license, Class I)
ICE386DP ICE386D	The complete ICE-386 emulator system including control unit, processor module, power supply, SAST, OIB, SAB, a serial communication cable (SCOM4), and software (S386DP/S386D). (Requires software license, Class I). Note: A GPIB adapter card and cable for the IBM PC AT is optionally available. Please contact your Intel Field Applications Engineer for more information.
SCOM4	A 12-foot serial cable for the ICE-386 — IBM PC AT connection
S386DP S386D	ICE-386 software for hosting on the IBM PC AT running DOS3.1
SCOM5	A 12-foot serial cable for the ICE-386 — Intel 286/310 connection
S386XP S386X	ICE-386 software for hosting on the Intel System 286/310 running XENIX release 3.0 with Update 3

ICE™-5100/044 In-Circuit Emulator for the RUPI™-44 Family

- Precise, Full-Speed, Real-Time Emulation of the RUPI™-44 Family of Peripherals
- 64 KB of Mappable High-Speed Emulation Memory
- 254 24-bit Frames of Trace Memory (16 Bits Trace Program Execution Addresses and 8 Bits Trace Eternal Events)
- Serial Link to Intel Series III/IV or IBM* PC AT or PC XT (and PC DOS Compatibles)
- ASM-51 and PL/M-51 Language Support
- Built-in CRT-Oriented Text Editor
- Symbolic Debugging Enables Access to Memory Locations and Program Variables
- Four Address Breakpoints Plus In-Range, Out-of-Range, and Page Breaks
- Equipped with the Integrated Command Directory (ICD™) That Provides
  - On-Line Help
  - Syntax Guidance and Checking
  - Command Recall
- On-Line Disassembler and Single-Line Assembler to Help with Code Patching
- Provides an Ideal Environment for Debugging BITBUS™ Applications Code

The ICE™-5100/044 in-circuit emulator is a high-level, interactive debugger that is used to develop and test the hardware and software of a target system based on the RUPI™-44 family of peripherals. The ICE-5100/044 emulator can be serially linked to an Intellec® Series III/IV or an IBM PC AT or PC XT. The emulator can communicate with the host system at standard baud rates up to 19.2K. The design of the emulator supports all of the RUPI-44 components at speeds up to and including 12 MHz.



280325-1

November 1986
Order Number: 280325-001

PRODUCT OVERVIEW

The ICE-5100/044 emulator provides full emulation support for the RUPI-44 family of peripherals, including 8044-based BITBUS™ board products. The RUPI-44 family consists of the 8044, the 8744, and the 8344.

The ICE-5100/044 emulator enables hardware and software development to proceed simultaneously. With the ICE-5100/044, prototype hardware can be added to the system as it is designed and software can be developed prior to the completion of the hardware prototype. Software and hardware integration can occur while the product is being developed.

The ICE-5100/044 emulator assists four stages of development:

- Software debugging
- Hardware debugging
- System integration
- System test

Software Debugging

The ICE-5100/044 emulator can be operated without being connected to the target system and before any of the user's hardware is available (provided external data RAM is not needed). In this stand-alone mode, the ICE-5100/044 emulator can be used to facilitate program development.

Hardware Debugging

The ICE-5100/044 emulator's AC/DC parametric characteristics match the microcontroller's. The emulator's full-speed operation makes it a valuable tool for debugging hardware, including time-critical serial port, timer, and external interrupt interfaces.

System Integration

Integration of software and hardware can begin when the emulator is plugged into the microcontroller socket of the prototype system hardware. Hardware can be added, modified, and tested immediately. As each section of the user's hardware is completed, it can be added to the prototype. Thus, the hardware and software can be system tested in real-time operation as each section becomes available.

System Test

When the prototype is complete, it is tested with the final version of the system software. The ICE-5100/044 emulator is then used for real-time emulation of the microcontroller to debug the system as a completed unit.

The final product verfication test can be performed using the ROM or EPROM version of the microcontroller. Thus, the ICE-5100/044 emulator provides the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

PHYSICAL DESCRIPTION

The ICE-5100/044 emulator consists of the following components (see Figure 1):

- Power supply
- AC and DC power cables
- Controller pod
- Serial Cable (host-specific)
- User probe assembly (consisting of the processor module and the user cable)
- Crystal power accessory (CPA)
- 40-pin target adaptor
- Clips assembly
- Software (includes the ICE-5100/044 emulator software, diagnostic software, and a tutorial)

The controller pod contains 64 KB of emulation memory, 254- by 24-bit frames of trace memory, and the control processor. In addition, the controller pod houses a BNC connector that can be used to connect up to 10 multi-ICE compatible emulators for synchronous starting and stopping of emulation.

The serial cable connects the host system to the controller pod. The serial cable supports a subset of the RS-232C signals.

The user probe assembly consists of a user cable and a processor module. The processor module houses the emulation processor and the interface logic. The target adaptor connects to the processor module and provides an electrical and mechanical interface to the target microcontroller socket.

The crystal power accessory (CPA) is a small, detachable board that connects to the controller pod and enables the ICE-5100/044 emulator to run in stand-alone mode. The target adaptor plugs into the socket on the CPA; the CPA then supplies clock and power to the user probe.

The clips assembly enables the user to trace external events. Eight bits of data are gathered on the rising edge of $\overline{\text{PSEN}}$ during opcode fetches. The clips information can be displayed using the CLIPS option with the PRINT command.

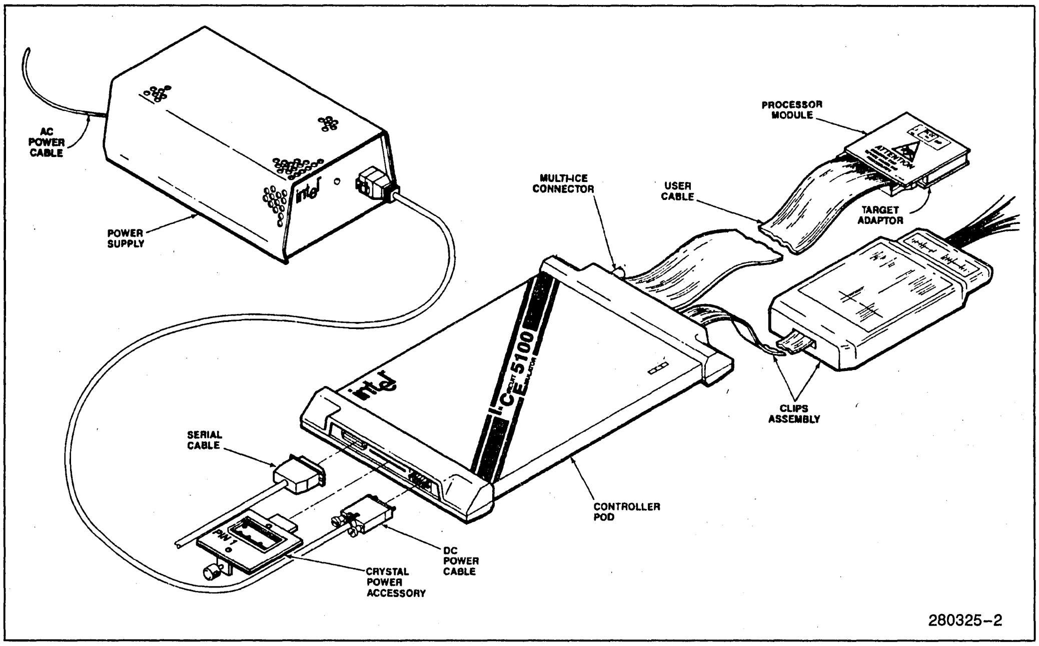


Figure 1. The ICE™-5100/044 Emulator Hardware

The ICE-5100-044 emulator software supports mnemonics, object file formats, and symbolic references generated by Intel's ASM-51 and PL/M-51 programming languages. Along with the ICE-5100/044 emulator software is a customer confidence test disk with diagnostic routines that check the operation of the hardware.

The on-line tutorial is written in the ICE-5100 command language. Thus, the user is able to interact with and use the ICE-5100/044 emulator while executing the tutorial.

A comprehensive set of documentation is provided with the ICE-5100/044 emulator.

ICE™-5100/044 EMULATOR FEATURES

The ICE-5100/044 emulator has been created to assist a product designer in developing, debugging and testing designs incorporating the RUPI-44 family of peripherals. The following sections detail some of the ICE-5100/044 emulator features.

Emulation

Emulation is the controlled execution of the user's software in the target hardware or in an artificial hardware environment that duplicates the microcontroller of the target system. Emulation is a transparent process that happens in real-time. The execution of the user software is facilitated with the ICE-5100/044 command language.

Memory Mapping

There is a 64 KB of memory that can be mapped to the CODE memory space in 4 KB blocks on 4 KB boundaries. By mapping memory to the ICE-5100/044 emulator, software development can proceed before the user hardware is available.

Memory Examination and Modification

The memory space for the 8044 microcontroller and its target hardware is fully accessible through the emulator. The ICE-5100/044 emulator refers to four physically distinct memory spaces, as follows:

- CODE—references program memory
- IDATA—references internal data memory
- RDATA—references special function register memory
- XDATA—references external data memory

ICE-5100/044 emulator commands that access memory use one of the special prefixes (e.g., CODE) to specify the memory space.

The microcontroller's special function registers and register bits can be accessed mnemonically (e.g., DPL, TCON, CY, P1.2) with the ICE-5100/044 emulator software.

Data can be displayed or modified in one of three bases: hexadecimal, decimal, or binary. Data can also be displayed or modified in one of two formats: ASCII or unsigned integer. Program code can be disassembled and displayed as ASM-51 assembler mnemonics. Code can be modified with standard ASM-51 statements using the built-in single-line assembler.

Symbolic references can be used to specify memory locations. A symbolic reference is a procedure name, line number, program variable, or label in the user program that corresponds to a location.

Some typical symbolic functions include:

- Changing or inspecting the value of a program variable by using its symbolic name to access the memory location.
- Defining break and trace events using symbolic references.
- Referencing variables as primitive data types. The primitive data types are ADDRESS, BIT, BOOLEAN, BYTE, CHAR (character), and WORD.

The ICE-5100/044 emulator maintains a virtual symbol table (VST) for program symbols. A maximum of 61 KB of host memory space is available for the VST. If the VST is larger than 61 KB, the excess is stored on available host system disk space and is paged in and out as needed. The size of the VST is limited only by the disk capacity of the host system.

Breakpoint Specifications

Breakpoints are used to halt a user program in order to examine the effect of the program's execution on the target system. The ICE-5100/044 emulator supports three different types of break specifications:

- Specific address break—specifying a single address point at which emulation is to be stopped.
- Range break—an arbitrary range of addresses can be specified to halt emulation. Program execution within or, optionally, outside the range halts emulation.
- Page break—up to 256 page breaks can be specified. A page break is defined as a range of addresses that is 256-bytes long and begins on a 256-byte address boundary.

Break registers are user-defined debug definitions used to create and store breakpoint definitions. Break registers can contain multiple breakpoint definitions and can optionally call debug procedures when emulation halts.

Trace Specifications

Tracing can be triggered using specifications similar to those used for breaking. Normally, the ICE-5100/044 emulator traces program activity while the user program is executing. With a trace specification, tracing can be triggered to occur only when specific conditions are met during execution. Up to 254 24-bit frames of trace information are collected in a buffer during emulation. Sixteen of the 24 bits trace instruction execution addresses, and 8 bits capture external events (CLIPS).

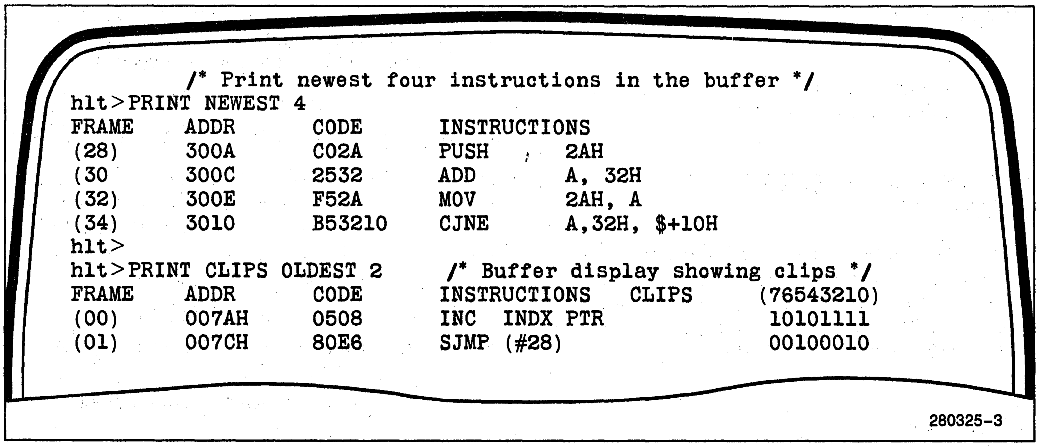
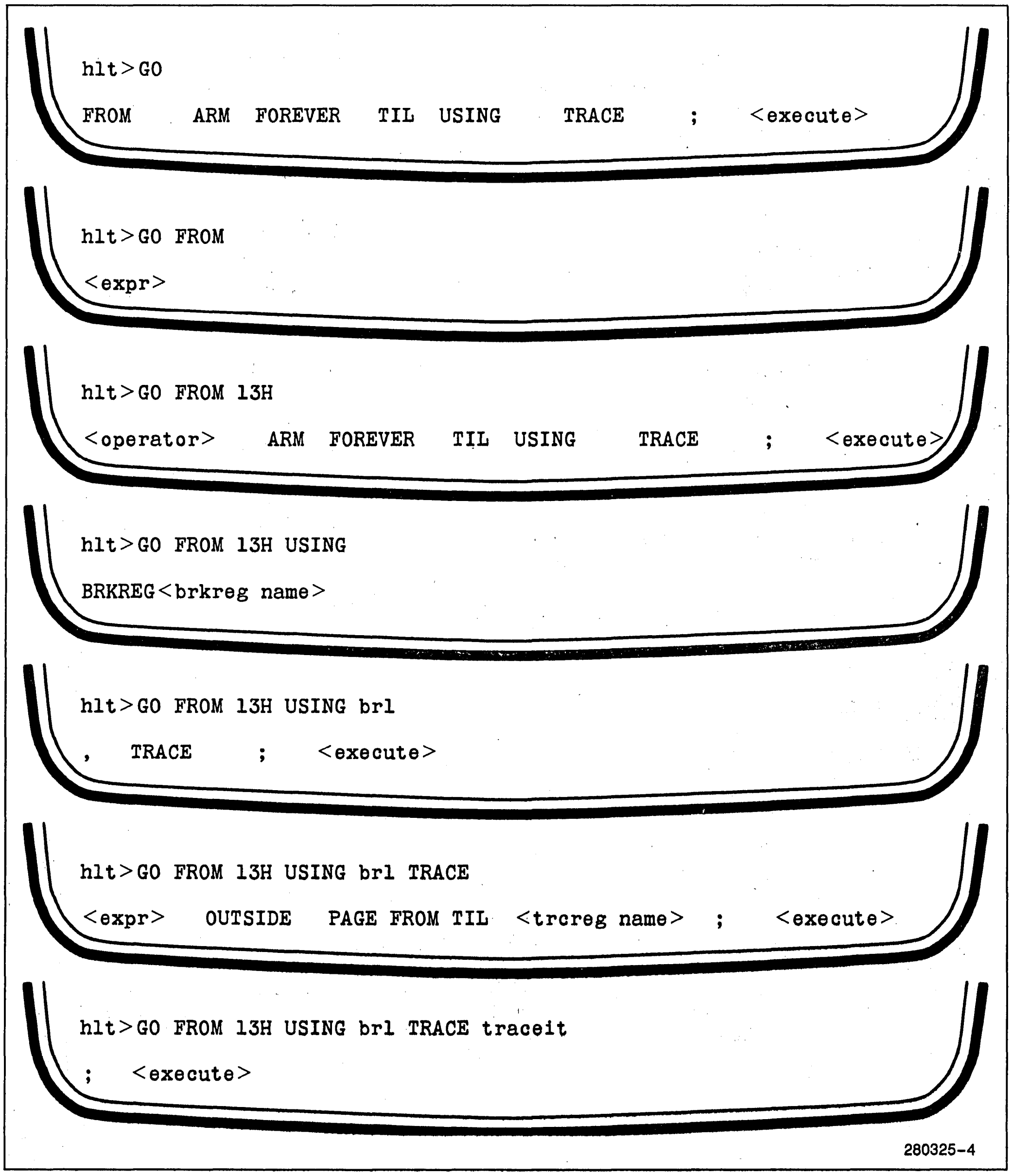

```
        /* Print newest four instructions in the buffer */
hlt>PRINT NEWEST 4
FRAME    ADDR      CODE       INSTRUCTIONS
(28)     300A      C02A       PUSH       2AH
(30      300C      2532       ADD        A, 32H
(32)     300E      F52A       MOV        2AH, A
(34)     3010      B53210     CJNE       A,32H, $+10H
hlt>
hlt>PRINT CLIPS OLDEST 2      /* Buffer display showing clips */
FRAME    ADDR      CODE       INSTRUCTIONS    CLIPS      (76543210)
(00)     007AH     0508       INC  INDX PTR               10101111
(01)     007CH     80E6       SJMP (#28)                  00100010
```
280325-3

Figure 2. Selected Trace Buffer Displays

The trace buffer display is similar to an ASM-51 program listing as shown in Figure 2. The PRINT command enables the user to selectively display the contents of the trace buffer. The user has the option of displaying the clips information as well as dissassembled instructions.

Procedures

Debugging procedures (PROCs) are a user-defined group of ICE-5100/044 commands that are executed as one command. PROCs enable the user to define several commands in a named block structure. The commands are executed by entering thename of the PROC. The PROC bodies are a simple DO . . . END construct.



```
hlt>GO
FROM   ARM  FOREVER   TIL  USING    TRACE     ;     <execute>

hlt>GO FROM
<expr>

hlt>GO FROM 13H
<operator>   ARM  FOREVER   TIL  USING    TRACE     ;     <execute>

hlt>GO FROM 13H USING
BRKREG<brkreg name>

hlt>GO FROM 13H USING brl
,   TRACE     ;    <execute>

hlt>GO FROM 13H USING brl TRACE
<expr>   OUTSIDE   PAGE FROM TIL  <trcreg name>  ;    <execute>

hlt>GO FROM 13H USING brl TRACE traceit
;   <execute>
```

280325-4

Figure 3. The Integrated Command Directory for the GO Command

PROCs can simulate missing hardware or software, set breakpoints, collect debug information, and execute high-level software patches. PROCs can be copied to text files on disk, then recalled for use in later test sessions. PROCs can also serve as program diagnostics, implementing ICE-5100/044 emulator commands or user-defined definitions for special purposes.

On-Line Syntax Menu

A special syntax menu, called the Integrated Command directory (ICD), similar to the one used for the I²ICE™ system and the VLSiCE-96 emulator, aids in creating syntactically correct command lines. Figure 3 shows an example of the ICD and how it changes to reflect the options available for the GO command.

Help

The HELP command provides ICE-5100/044 emulation command assistance via the host system terminal. On-line HELP is available for the ICE-5100/044 emulator commands shown in Figure 4.

BITBUS™ Applications Support

The ICE-5100/044 emulator provides an ideal environment for developing applications code for BITBUS board products such as the RCB-44/10, the RCB-44/20, the PCX-344, and the iSBX™-344 board.

The BITBUS firmware, available separately as BITWARE, can be loaded into the ICE-5100/044 emulator's memory along with the user's code to enable rapid debug of 8044 BITBUS applications code.

DESIGN CONSIDERATIONS

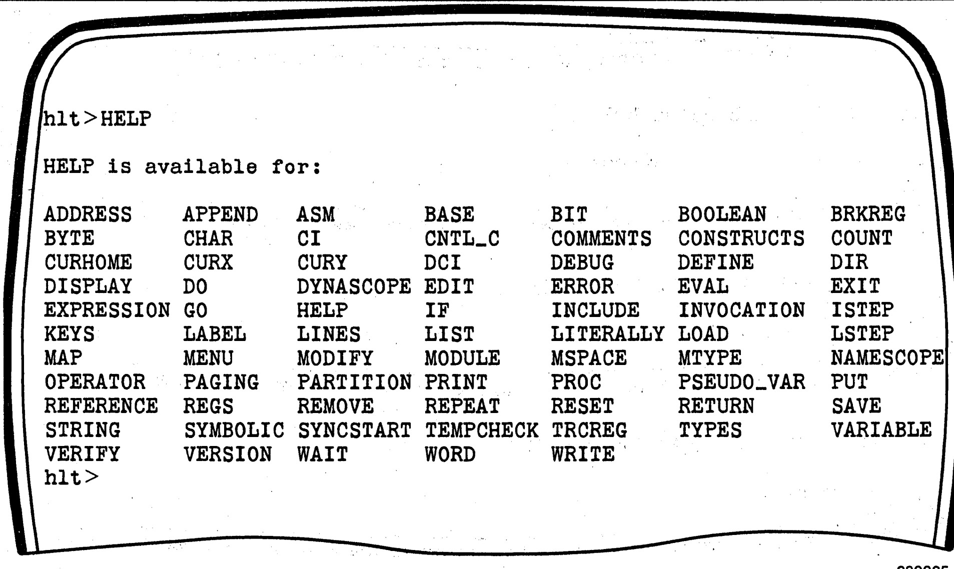
The height of the processor module and the target adaptor need to be considered for target systems. Allow at least 1½ inches (3.8 cm) of space to fit the processor module and target adaptor. Figure 5 shows the dimensions of the processor module.

Execution of user programs that contain interrupt routines causes incorrect data to be stored in the trace buffer. When an interrupt occurs, the next instruction to be executed is placed into the trace buffer before it is actually executed. Following completion of the interrupt routine, the instruction is executed and again placed into the trace buffer.

ELECTRICAL CONSIDERATIONS

The emulation processor's user-pin timings and loadings are identical to the 8044 component, except as follows.

- Up to 25 pF of additional pin capacitance is contributed by the processor module and target adaptor assemblies.
- Pin 31, $\overline{EA}$, has approximately 32 pF of additional capacitance loading due to sensing circuitry.
- Pins 18 and 19, XTAL1 and XTAL2 respectively, have approximately 15-16 pF of additional capacitance when configured for crystal operation.



```
hlt>HELP

HELP is available for:

ADDRESS     APPEND    ASM        BASE       BIT        BOOLEAN     BRKREG
BYTE        CHAR      CI         CNTL_C     COMMENTS   CONSTRUCTS  COUNT
CURHOME     CURX      CURY       DCI        DEBUG      DEFINE      DIR
DISPLAY     DO        DYNASCOPE  EDIT       ERROR      EVAL        EXIT
EXPRESSION  GO        HELP       IF         INCLUDE    INVOCATION  ISTEP
KEYS        LABEL     LINES      LIST       LITERALLY  LOAD        LSTEP
MAP         MENU      MODIFY     MODULE     MSPACE     MTYPE       NAMESCOPE
OPERATOR    PAGING    PARTITION  PRINT      PROC       PSEUDO_VAR  PUT
REFERENCE   REGS      REMOVE     REPEAT     RESET      RETURN      SAVE
STRING      SYMBOLIC  SYNCSTART  TEMPCHECK  TRCREG     TYPES       VARIABLE
VERIFY      VERSION   WAIT       WORD       WRITE
hlt>
```

280325-5

Figure 4. HELP Menu

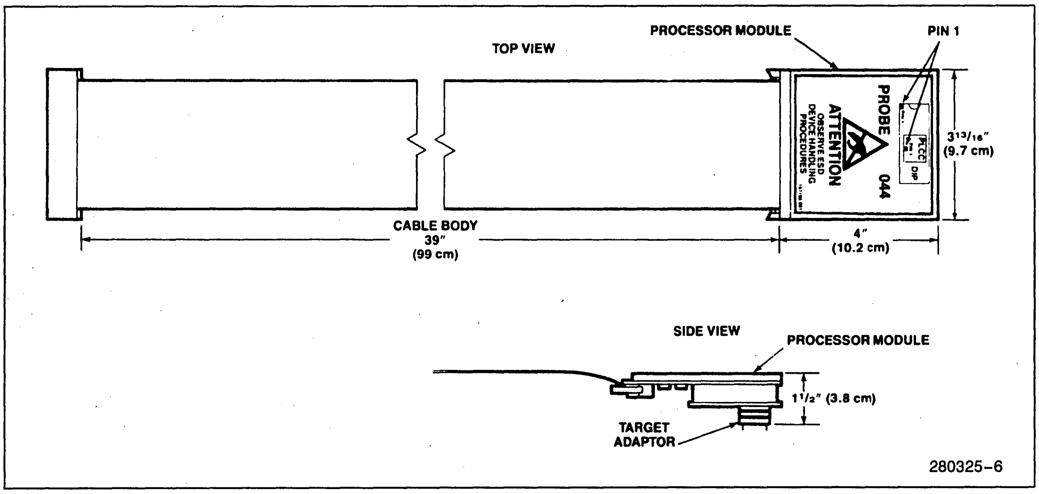


Figure 5. Processor Module Dimensions

HOST REQUIREMENTS

- IBM PC AT or PC XT (or PC DOS compatible) with 512 KB of available RAM and a hard disk running under the DOS 3.0 (or later) operating system.
- Intellec Series III/IV microcomputer development system running the ISIS or iNDX operating system respectively, with at least 512 KB of application memory resident.
- Disk drives—dual floppy or one hard disk and one floppy drive required.

ICE™-5100/044 EMULATOR SOFTWARE PACKAGE

- ICE-5100/044 emulator software
- ICE-5100/044 confidence tests
- ICE-5100 tutorial software

EMULATOR PERFORMANCE

Memory

Mappable full-speed emulation code memory	64 KB	Mappable to user or ICE-5100/044 emulator memory in 4 KB blocks on 4 KB boundaries.
Trace memory		254 x 24 bit frames
Virtual Symbol Table		A maximum of 61 KB of host memory space is available for the virtual symbol table (VST). The rest of the VST resides on disk and is paged in and out as needed.

PHYSICAL CHARACTERISTICS

Controller Pod

Width:	8-¼"	(21 cm)
Height:	1-½"	(3.8 cm)
Depth:	13-½"	(34.3 cm)
Weight:	4 lbs	(1.85 kg)

User Cable

The user cable is 3 feet (approximately 1 m)

Processor Module

(With the target adaptor attached)

Width:	3-¹³⁄₁₆"	(9.7 cm)
Height:	4"	(10.2 cm)
Depth:	1-½"	(3.8 cm)

Power Supply

Width:	7-⅝"	(18.1 cm)
Height:	4"	(10.06 cm)
Depth:	11"	(27.97 cm)
Weight	15 lbs	(6.1 kg)

Serial Cable

The serial cable is 12 feet (3.6 m).

ELECTRICAL CHARACTERISTICS

Power Supply

100-120V or 200-240V (selectable)
50-60 Hz
2 amps (AC max) @ 120V
1 amp (AC max) @ 240V

ENVIRONMENTAL CHARACTERISTICS

Operating temperature	+10°C to +40°C (50°F to 104°F)
Operating humidity	Maximum of 85% relative humidity, non-condensing

ORDERING INFORMATION

Emulator Hardware and Software

Order Code	Description
I044KITAD	This kit contains the ICE-5100/044 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with an IBM PC AT or PC XT. The kit also includes the 8051 Software Development Package and the AEDIT text editor for use on DOS systems. [Requires software license.]
I044KITD	This kit is the same as the I044KITAD excluding the 8051 Software Development Package and the AEDIT text editor. [Requires software license.]
I044KITAS	This kit contains the ICE-5100/044 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with Intel hosts (Series III/IV). The kit also includes the 8051 Software Development Package and the AEDIT text editor for use on the Series III/IV. [Requires software license.]
I044KITS	This kit is the same as the I044KITAS excluding the 8051 Software Development Package and the AEDIT text editor. [Requires software license.]

Software Only

Order Code	Description
SA044D	This kit contains the host, probe, diagnostic, and tutorial software on 5¼" disks for use on an IBM PC AT or PC XT (requires DOS 3.0 or later). [Requires software license.]
SA044S	This kit contains the host, probe, diagnostic, and tutorial software on 8" disks (both single-density and double-density) for use on a Series III, and on 5-¼" disks for use on a Series IV. [Requires software license.]

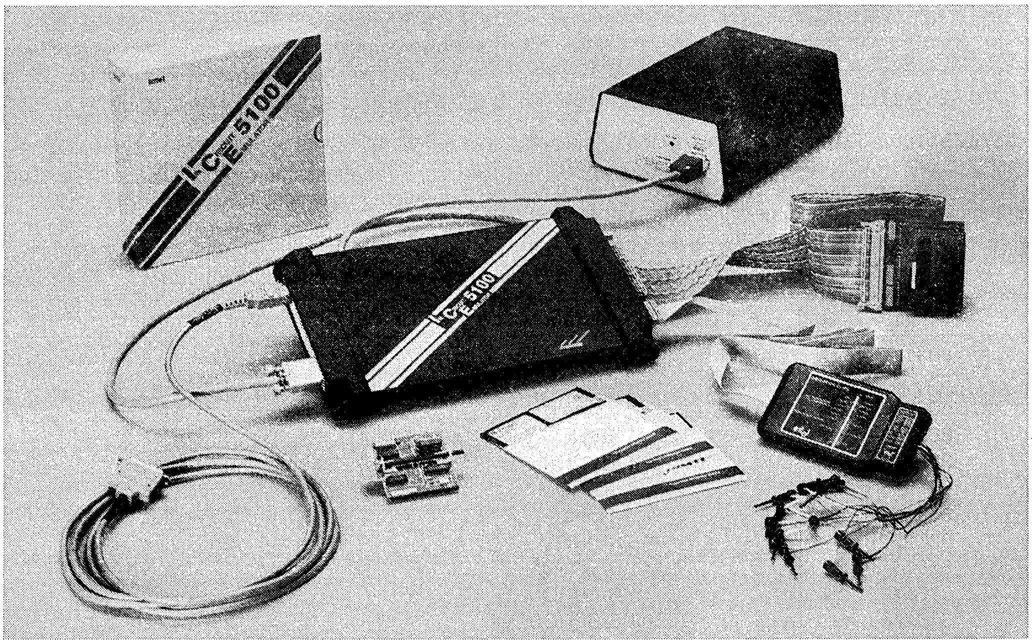
Other Useful Intel® MCS®-51 Debug and Development Support Products

Order Code	Description
D86ASM51	**8051 Software Development Package** (DOS version)—Consists of the ASM-51 macro assembler which gives symbolic access to 8051 hardware features; the RL51 linker and relocator program that links modules generated by ASM-51; CONV51 which enables software written for the MCS-48 family to be up-graded to run on the 8051, and the LIB51 Librarian which programmers can use to create and maintain libraries of software object modules. Use with the DOS operating system (version 3.0 or later).
D86PLM51	**PL/M-51 Software Package** (DOS version)—Consists of the PL/M-51 compiler which provides high-level programming language support; the LIB51 utility that creates and maintains libraries of software object modules, and the RL51 linker and relocator program that links modules generated by ASM-51 and PL/M-51 and locates the linked object modules to absolute memory locations. Use with the DOS operating system (version 3.0 or later).
I86ASM51	**8051 Software Development Package** (ISIS version)—Same as the D86ASM51 package except this one is for use with the Series III.
I86PLM51	**PL/M-51 Sofware Package**—Same as the D86PLM51 package except this one is for use with the Series III and Series IV.
D86EDINL	AEDIT text editor for use with the DOS operating system.

ICE™-5100/252 In-Circuit Emulator for the MCS®-51 Family of Microcontrollers

- **Real-time Emulation of Selected MCS®-51 Microcontroller Components at Speeds up to 16 MHz**
- **64 KB of Mappable High-Speed Emulation Memory**
- **254 24-Bit Frames of Trace Memory (16 Bits Trace Program Execution Addresses and 8 Bits Trace External Events)**
- **Serial Link to Intel Series III/IV or IBM* PC AT and PC XT (and PC-DOS Compatibles)**
- **ASM-51 and PL/M-51 Language Support**
- **Built-In CRT-Oriented Text Editor**
- **Symbolic Debugging Enables Access to Memory Locations and Program Variables**
- **Four Address Breakpoints Plus In-Range, Out-of-Range, and Page Breaks**
- **Equipped with the Integrated Command Directory (ICD™) that Provides**
  - **—On-Line Help**
  - **—Syntax Guidance and Checking**
  - **—Dynamic Command-Entry**
  - **—Error Checking**
  - **—Command Recall**
- **On-Line Disassembler and Single-Line Assembler to Help with Code Patching**

The ICE™-5100/252 In-Circuit Emulator is a high-level, interactive debugger that is used to develop and test the hardware and software of a target system based on the MCS-51 family of microcontrollers. The ICE-5100/252 emulator can be serially linked to an Intellec® Series III/IV or an IBM PC AT or PC XT. The emulator can communicate with the host system at standard baud rates up to 19.2K. The design of the emulator supports selected MCS-51 microcontroller components at speeds up to 16 MHz.



280200-1

*IBM is a registered trademark of International Business Machines Corporation.

November 1986
Order Number: 280200-001

PRODUCT OVERVIEW

The ICE-5100/252 emulator provides full emulation support for the MCS-51 family members listed in Table 1.

The ICE-5100/252 emulator enables hardware and software development to proceed simultaneously. With the ICE-5100/252 emulator, prototype hardware can be added to the system as it is designed and software can be developed prior to the completion of the hardware prototype. Software and hardware integration can occur while the product is being developed.

The ICE-5100/252 emulator assists four stages of development:

- Software debugging
- Hardware debugging
- System integration
- System test

Software Debugging

The ICE-5100/252 emulator can be operated without being connected to the target system or before any of the user's hardware is available (provided external data RAM is not needed). In this stand-alone mode, the ICE-5100/252 emulator can be used to facilitate program development.

Hardware Debugging

The ICE-5100/252 emulator's AC/DC parametric characteristics match the microcontroller's; its full-speed operation makes it a valuable tool for debugging hardware, including time-critical serial port, timer, and external interrupt interfaces.

System Integration

Integration of software and hardware can begin when the emulator is plugged into the microcontroller socket of the prototype system hardware. Hardware can be added, modified, and tested immediately. As each section of the user's hardware is completed, it can be added to the prototype. Thus, the hardware and software can be system tested in real-time operation as it becomes available.

System Test

When the prototype is complete, it is tested with the final version of the system software. The ICE-5100/252 emulator is then used for real-time emulation of the microcontroller to debug the system as a completed unit.

The final product verification test can be performed using the ROM orEPROM version of the microcontroller. Thus, the ICE-5100/252 emulator provides the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

PHYSICAL DESCRIPTION

The ICE-5100/252 emulator consists of the following components (see Figure 1):

- Power supply
- AC and DC power cables
- Controller pod
- Serial cable (host-specific)
- User probe assembly (consisting of processor module and user cable)
- Crystal power accessory (CPA)

Table 1. MCS®-51 Family Support Offered by the ICE™-5100/252 Emulator

Part	On-Chip Program Memory	On-Chip Data Memory
8031	None	128 bytes
80C31	None	128 bytes
8032	None	256 bytes
8051	4 KB-ROM	128 bytes
80C51	4 KB-ROM	128 bytes
8052	8 KB-ROM	256 bytes
80C252	None	256 bytes
83C252	8 KB-ROM	256 bytes
8751	4 KB-EPROM	128 bytes
87C51	4 KB-EPROM	128 bytes
8752	8 KB-EPROM	256 bytes
87C252	8 KB-EPROM	256 bytes

- 40-pin DIP target adaptor
- Clips assembly
- Software (includes the ICE-5100/252 emulator software, diagnostic software, and tutorial).

The controller pod contains 64 KB of emulation memory, a 254-frame trace buffer, and the control processor. In addition, the controller pod houses a BNC connector that can be used to connect up to 10 multi-ICE compatible systems together for synchronous GO and BREAK emulation.

The serial cable connects the host system to the controller pod. The serial cable supports a subset of the RS-232C signals.

The user probe assembly consists of a user cable and a processor module. The processor module houses the emulation processor and provides the logic needed to support mapped memory, breakpoints, emulation, interrogation, and modification of registers and memory. The target adaptor connects to the processor module and provides an electrical and mechanical interface to the target microcontroller socket.

The crystal power accessory (CPA) is a small, detachable board that connects to the controller pod and enables you to run the ICE-5100/252 emulator in a stand-alone (loop-back) mode of operation. In stand-alone mode, the target adaptor plugs into the socket on the CPA; the CPA then supplies clock and power to the user probe.

The clips assembly enables the user to trace external events. Eight bits of data are gathered on the rising edge of $\overline{\text{PSEN}}$ during opcode fetches. The clips information can be displayed using the CLIPS option with the PRINT command.

The ICE-5100/252 emulator software supports mnemonics, object file formats, and symbolic references generated by Intel's ASM-51 and PL/M-51 programming languages. Along with the ICE-5100/252 emulator software is a customer confidence test disk with diagnostic routines that check the operation of the hardware.

The on-line tutorial is written in the ICE-5100/252 command language. Thus, the user is able to interact with and use the ICE-5100/252 emulator while executing the tutorial.

A comprehensive set of documentation is included with the ICE-5100/252 emulator.

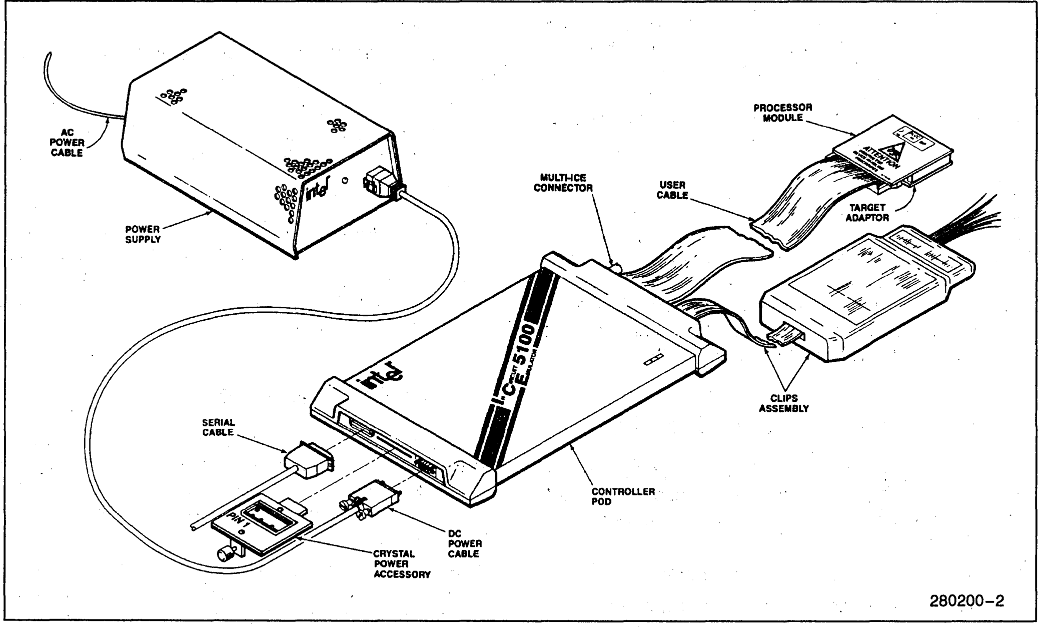


Figure 1. The ICE™-5100/252 Emulator Hardware

ICE™-5100/252 EMULATOR FEATURES

The ICE-5100/252 emulator has been created to assist a product designer in developing, debugging and testing designs incorporating the MCS-51 family of microcontrollers. The following sections detail some of the ICE-5100/252 emulator features.

Processor Selection

The ICE-5100/252 emulator enables you to emulate the microcontrollers listed in Table 1. Selecting a processor type changes the following characteristics to match the microcontroller selected:

- Internal RAM size
- Internal ROM size
- Idle and power down mode enable
- Special function register symbolic map
- Memory map
- Latched or unlatched $\overline{EA}$
- Serial port framing error detection

Emulation

Emulation is the controlled execution of the user's software in the target hardware or in an artificial hardware environment that duplicates the microcontroller of the target system. Emulation is a transparent process that happens in real-time. The execution of the user software is facilitated through the ICE-5100/252 command language.

Memory Mapping

There is 64 KB of memory that can be mapped to the CODE memory space in 4 KB blocks on 4K boundaries. By mapping memory to the ICE-5100/252 emulator, software development can proceed before user hardware is available.

Memory Examination and Modification

The memory space for the MCS-51 component(s) and its target hardware is fully accessible through the emulator. The microcontroller has four physically distinct memory spaces:

- CODE — references program memory
- IDATA — references internal data memory
- RDATA — references special function register memory
- XDATA — references external data memory

ICE-5100/252 emulator commands that access memory must use one of the special prefixes (e.g., CODE) to specify the memory space in which the partition lies.

The microcontroller's special function registers and register bits can be accessed mnemonically (e.g., DPL, TCON, CY) with the ICE-5100/252 emulator software.

Data can be displayed or modified in one of three bases:hexadecimal, decimal, or binary and in ASCII and unsigned integer formats. Program code can be disassembled and displayed as ASM-51 assembler mnemonics. Code can be modified with standard ASM-51 statements using the built-in single-line assembler.

Symbolic debugging is used to specify memory locations by their symbolic references. A symbolic reference is a procedure name, line number, or label in the user program that corresponds to a location. Using symbolics to reference program locations is a mnemonic way of accessing the program.

Some typical symbolic functions include:

- Changing or inspecting the value of a program variable by using its symbolic name to access the memory location.
- Defining break and trace events using symbolic references.
- Referencing variables as primitive data types. The primitive data types are ADDRESS, BIT, BOOLEAN, BYTE, CHAR (character), and WORD.

The ICE-5100/252 emulator maintains a virtual symbol table for program symbols making it possible for the table to exist without fitting entirely into host RAM memory. The size of the table is constrained only by the disk capacity.

Breakpoint Specifications

Breakpoints are used to halt a user program in order to examine the effect of the program's execution on the target system. The ICE-5100/252 emulator supports three different types of break specifications in real-time mode:

- Specific address break — Specifying a single address point at which emulation is to be stopped. This address can be an executable program statement or a program label.

- Range break — An arbitrary range of addresses can be specified to halt emulation. Program execution within or outside the range halts emulation.
- Page break — Up to 256 page breaks can be specified. A page break is defined as a range of addresses that is 256-bytes long and begins on a 256-byte address boundary.

Break registers are user-defined debug definitions used to create and store breakpoint definitions. Break registers can contain multiple breakpoint definitions and can optionally call debug procedures when emulation halts.

Trace Specifications

Tracing can be triggered using specifications similar to those used for breaking. Normally, the ICE-5100/252 emulator traces program activity while the user program is executing. With a trace specification, tracing can be triggered to occur only when specific conditions are met during execution. Up to 254 24-bit frames of trace information are collected in a buffer during emulation. Sixteen of the 24 bits trace instruction execution addresses, and 8 bits capture external events (CLIPS).

The trace buffer display is similar to an ASM-51 program listing as shown in Figure 2. The PRINT command enables the user to selectively display the contents of the trace buffer. The user has the option of displaying the clips information as well as disassembled instructions.

Procedures

Debugging procedures (PROCS) are a user-named group of ICE-5100/252 commands that are executed sequentially. PROCs can simulate missing hardware or software, collect debug information, execute high-level software patches, or make troubleshooting decisions. PROCs can be copied to text files on disk, then included from the file into the command sequence in later test sessions.

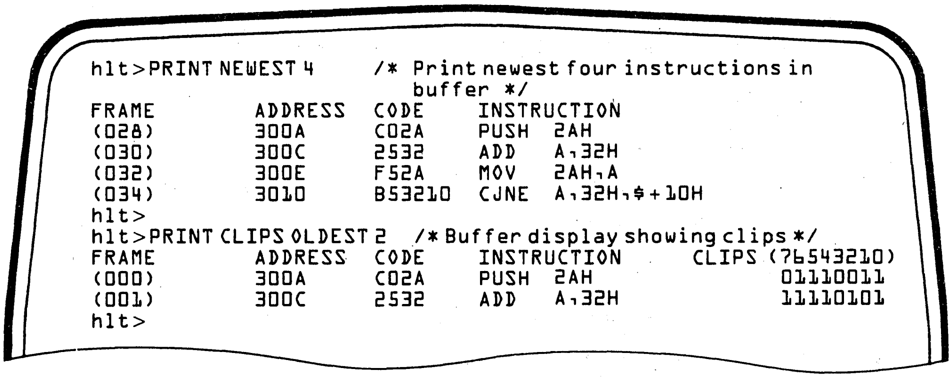
PROCs can also serve as programmable diagnostics, implementing ICE-5100/252 emulator commands or user-defined definitions for special purposes.

On-Line Syntax Menu

A special syntax menu, called the Integrated Command Directory (ICD), aids in creating syntactically correct command lines. Figure 3 shows an example of the ICD and how it changes to reflect the options available for the GO command.

HELP

The HELP command provides assistance with ICE-5100/252 emulation commands through the host system terminal. On-line HELP is available for the ICE-5100/252 emulator commands shown in Figure 4.



```
hlt>PRINT NEWEST 4      /* Print newest four instructions in
                           buffer */
FRAME       ADDRESS   CODE     INSTRUCTION
(028)       300A      C02A     PUSH  2AH
(030)       300C      2532     ADD   A,32H
(032)       300E      F52A     MOV   2AH,A
(034)       3010      B53210   CJNE  A,32H,$+10H
hlt>
hlt>PRINT CLIPS OLDEST 2  /* Buffer display showing clips */
FRAME       ADDRESS   CODE     INSTRUCTION      CLIPS (76543210)
(000)       300A      C02A     PUSH  2AH                01110011
(001)       300C      2532     ADD   A,32H              11110101
hlt>
```

280200–3

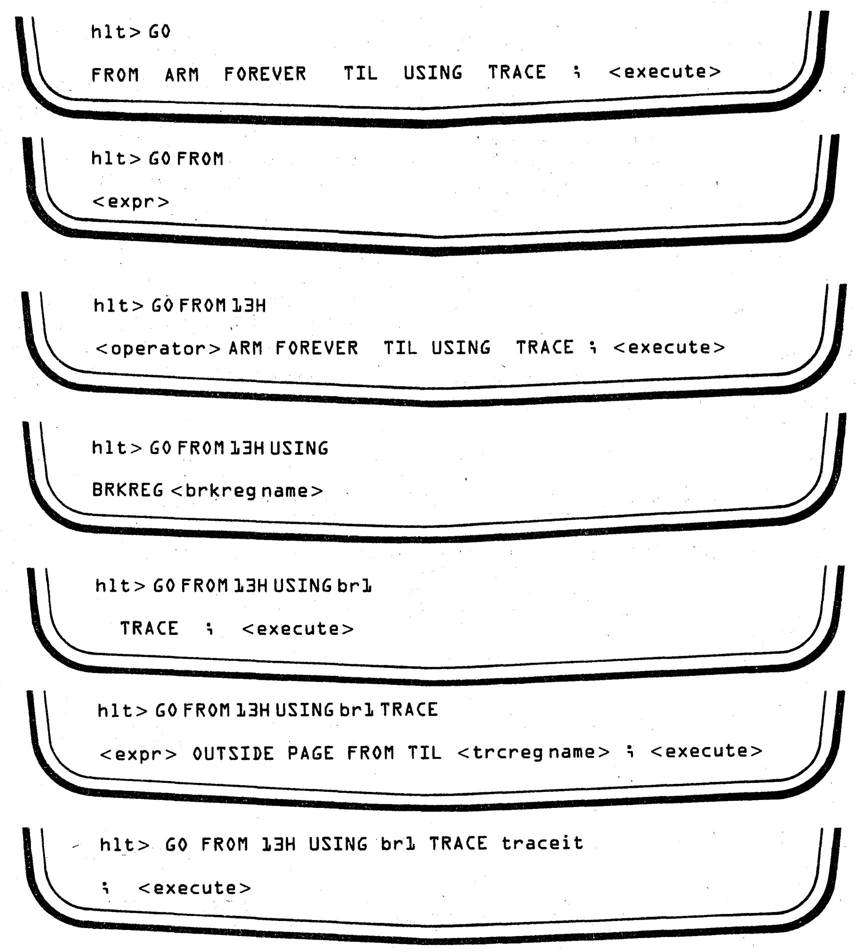
Figure 2. Selected Trace Buffer Displays

DESIGN CONSIDERATIONS

The height of the processor module and the target adaptor may pose a problem for multiple board target systems that need to be debugged. Allow at least 1½ inches (3.8 cm) of space between boards to fit the processor module and target adaptor. Figure 5 shows the dimensions of the processor module.

The following are limitations of the A-step emulation processor and should be kept in mind when using the ICE 5100 emulator. These problems will be fixed with the B-step version.

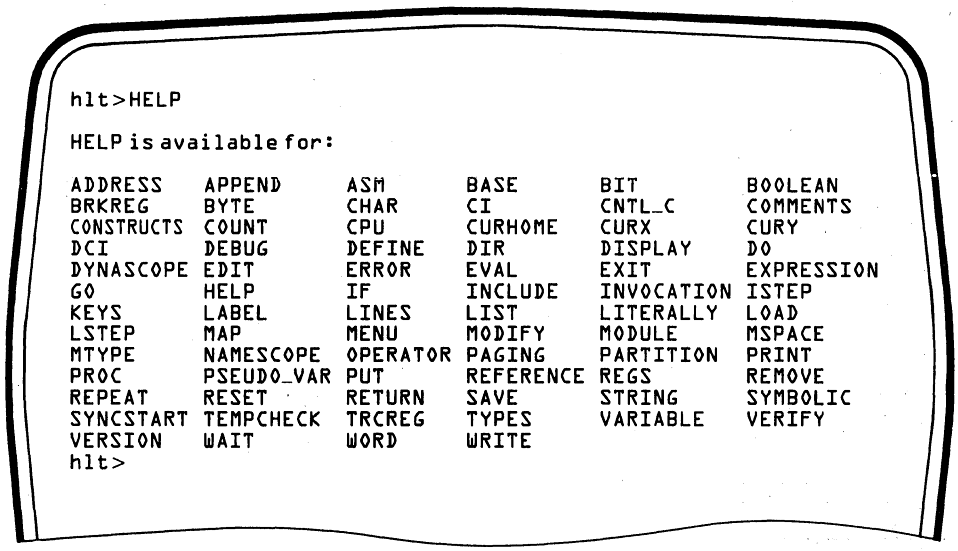
- The stack pointer of the emulation processor does not operate properly when pointing to addresses beyond 07FH. Internal data memory above 07FH can be addressed using standard indirect instructions.
- Execution of user programs that contain interrupt routines will cause incorrect data to be stored in the trace buffer. When an interrupt occurs, the next instruction to be executed is placed into the trace buffer before it is actually executed. Following completion of the interrupt routine, the instruction is executed and again placed into the trace buffer. There is no workaround for this bug at this time.



```
hlt> GO
FROM  ARM  FOREVER  TIL  USING  TRACE  ;  <execute>

hlt> GO FROM
<expr>

hlt> GO FROM 13H
<operator> ARM FOREVER  TIL USING  TRACE ; <execute>

hlt> GO FROM 13H USING
BRKREG <brkreg name>

hlt> GO FROM 13H USING br1
  TRACE  ;  <execute>

hlt> GO FROM 13H USING br1 TRACE
<expr> OUTSIDE PAGE FROM TIL <trcreg name> ; <execute>

hlt> GO FROM 13H USING br1 TRACE traceit
;  <execute>
```

280200–4

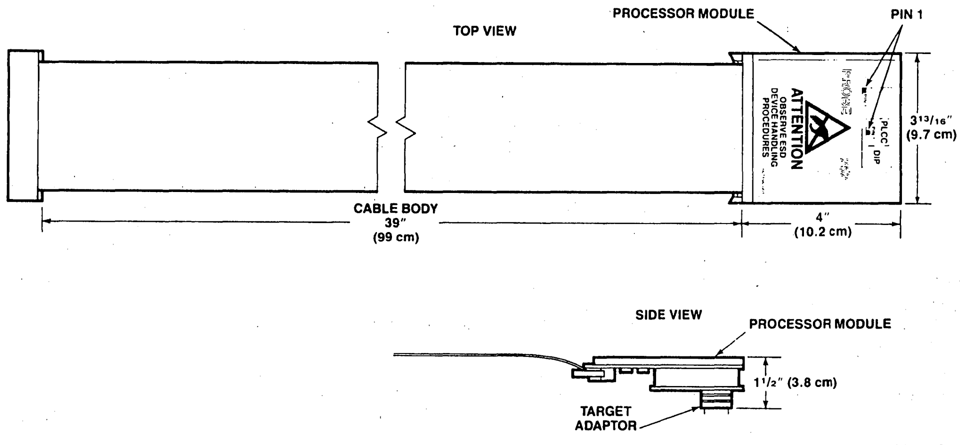
Figure 3. The Integrated Command Directory for the GO Command



```
hlt>HELP
HELP is available for:
ADDRESS    APPEND     ASM       BASE      BIT        BOOLEAN
BRKREG     BYTE       CHAR      CI        CNTL_C     COMMENTS
CONSTRUCTS COUNT      CPU       CURHOME   CURX       CURY
DCI        DEBUG      DEFINE    DIR       DISPLAY    DO
DYNASCOPE  EDIT       ERROR     EVAL      EXIT       EXPRESSION
GO         HELP       IF        INCLUDE   INVOCATION ISTEP
KEYS       LABEL      LINES     LIST      LITERALLY  LOAD
LSTEP      MAP        MENU      MODIFY    MODULE     MSPACE
MTYPE      NAMESCOPE  OPERATOR  PAGING    PARTITION  PRINT
PROC       PSEUDO_VAR PUT       REFERENCE REGS       REMOVE
REPEAT     RESET      RETURN    SAVE      STRING     SYMBOLIC
SYNCSTART  TEMPCHECK  TRCREG    TYPES     VARIABLE   VERIFY
VERSION    WAIT       WORD      WRITE
hlt>
```

280200-5

Figure 4. HELP Menu



280200-6

Figure 5. Processor Module Dimensions

ELECTRICAL CONSIDERATIONS

The emulation processor's user-pin timings and loadings are identical to the 80C252 component except as follows.

Maximum Operating ICC (ma)*

V_{CC}	4V	5V	6V
Frequency			
0.5 MHz	2.4	3.3	4.5
3.5 MHz	6.5	8.5	11.0
8.0 MHz	13.0	17.0	21.0
12.0 MHz	18.0	24.0	30.0
16.0 MHz	23.0	31.0	39.0

*ICC is measured with all output pins disconnected. XTAL1 driven with TCLCH, TCHCL = 10 ns, $V_{il} = V_{SS} + .5V$, $V_{ih} = V_{CC} - .5V$. XTAL2 not connected. $\overline{EA} = \overline{RST}$ = Port0 = V_{CC}.

Maximum Idle ICC (ma)*

VCC	4V	5V	6V
Frequency			
0.5 MHz	0.9	1.4	1.8
3.5 MHz	1.6	2.4	3.3
8.0 MHz	2.7	4.1	5.5
12.0 MHz	3.7	5.6	7.5
16.0 MHz	4.7	7.1	9.5

*Idle ICC is measured with all output pins disconnected. XTAL1 driven with TCLCH, TCHCL = 10ns, V_{il} = VSS + .5V, $V_{ih} = V_{CC} - .5V$. XTAL2 not connected. $\overline{EA}$ = PORT0 = V_{CC}, RST = V_{CC}, internal clock to PCA gated off.

- Up to 25 pf of additional pin capacitance is contributed by the processor module and target adaptor assemblies.
- Pin 31, $\overline{EA}$, has approximately 32 pf of additional capacitance loading due to sensing circuitry.
- Pins 18 and 19, XTAL1 and XTAL2 respectively, have approximately 15-16 pf of additional capacitance when configured for crystal operation.

Emulating HMOS Components

The ICE-5100/252 emulator is based on a CHMOS emulation processor. There are minor differences between how the ICE-5100/252 emulator supports CHMOS and HMOS designs as shown in Table 2.

Refer to the Mirocontroller Handbook, order number 210918, for further information on CHMOS and HMOS design considerations.

HOST REQUIREMENTS

- IBM PC AT or PC XT (or PC-DOS compatible) with 512 KB of RAM and a hard disk running under the DOS 3.0 (or later) operating system.
- Intellec Series III/IV Microcomputer Development System running under the ISIS or iNDX operating system respectively, with at least 512 KB of application memory resident.

  Disk drives — Dual floppy or one hard disk and one floppy drive required.

ICE-252 SYSTEM SOFTWARE PACKAGE

- ICE-5100/252 emulator software
- ICE-5100/252 confidence tests
- ICE-5100/252 tutorial software

SYSTEM PERFORMANCE

Memory

Mappable high-speed emulation code memory	Min 0 KB Max 64 KB	Mappable to user or ICE-5100/252 emulator memory in 4 KB blocks on 4 KB boundaries.
Trace Buffer	254 x 24 bits frames	

Table 2. CHMOS and HMOS Design Differences

Chip Function	HMOS Component 8031	CHMOS Component 80C31
RST trigger threshold	2.5V	70% Vcc (3.5V @ Vcc = 5V)
RST input impedance	4K − 10K ohms	50K − 150K ohms
Port I_{il}	−800μA	−50 μA
Clock threshold	2.5V	70% Vcc (3.5V @ Vcc = 5V)

Virtual Symbol Table	A maximum of 61 KB of host memory space is available for the virtual symbol table (VST). The rest of the VST resides on disk and is paged in and out as needed.

PHYSICAL CHARACTERISTICS

Controller Pod

Width	8¼" (21 cm)
Height	1½" (3.8 cm)
Depth	13½" (34.3 cm)
Weight	4 lbs (1.85 kg)

User Cable

3′ (.944 m)

Processor Module

(with target adaptor attached)

Width	3¹³⁄₁₆" (9.7 cm)
Length	4" (10.2 cm)
Height	1½" (3.8 cm)

Power Supply

Width	7⅝" (18.1 cm)
Height	4" (10.06 cm)
Depth	11" (27.97 cm)
Weight	15 lbs (6.1 kg)

Serial Cable

12′ (3.6 m)

ELECTRICAL CHARACTERISTICS

Power Supply

100 - 120V or 200 - 240V (selectable)
50 - 60 Hz
2 amps (AC max) - 120V
1 amp (AC max) - 240V

ENVIRONMENTAL CHARACTERISTICS

Operating temperature	+10° C to +40°C (37.5°F to 104°F)
Operating Humidity	Maximum of 85% relative humidity, non-condensing

ORDERING INFORMATION

Emulator Hardware and Software

Order Code	Description
I252KITAD	Consists of: ICE-5100/252 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with an IBM PC AT or PC XT. Kit also includes the 8051 Software Development Package and the AEDIT text editor for use on DOS systems. [Requires software license.]
I252KITD	Same as the I252KITAD package except this one does not include the 8051 Software Development Package or AEDIT text editor. [Requires software license.]
I252KITAS	Consists of: ICE-5100/252 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with Intel hosts (Series III, IV). Kit also includes the 8051 Software Development Package and the AEDIT text editor for use on Series III and Series IV. [Requires software license.]
I252KITS	Same as the I252KITAS package except this one does not include the 8051 Software Development Package or AEDIT text editor. [Requires software license.]

Software Only

Order Code	Description
SA252D	Kit contains the software for the host, probe, diagnostic, and tutorial on 5¼-inch disks for use on an IBM PC AT or PC XT (requires DOS 3.0 or later). [Requires software license].

SA252S	Kit contains the software for the host, probe, diagnostic, and tutorial on 8-inch disks (both single-density and double-density) for use on a Series III, and on 5¼-inch disks for use on a Series IV. [Requires software license].

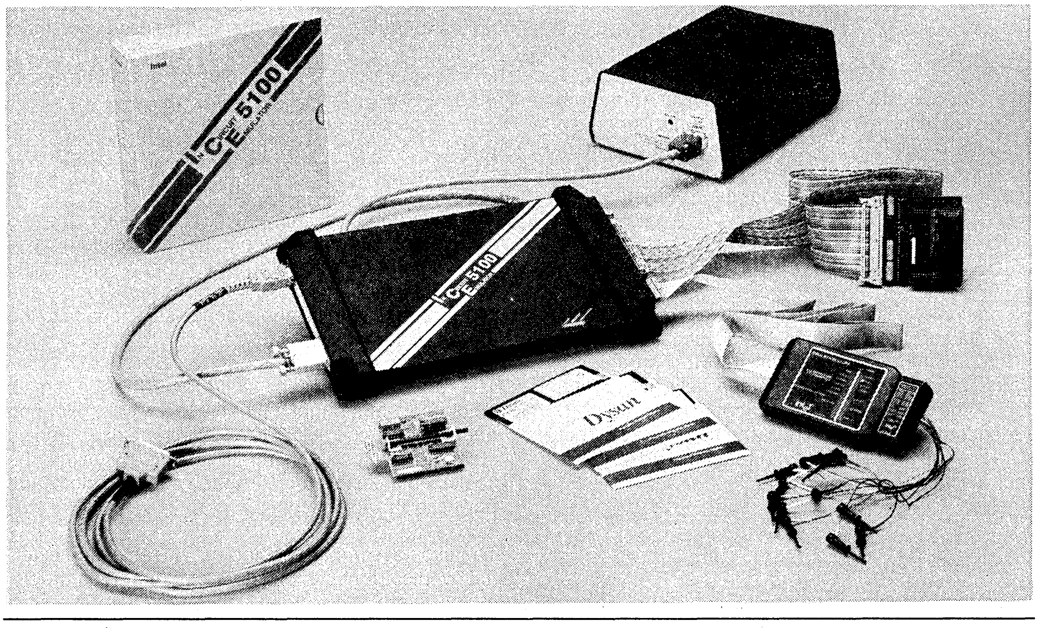
Other Useful Intel MCS-51 Debug and Development Support Products

Order Code	Description
D86ASM51	8051 Software Development Package (DOS version) — Consists of the ASM-51 macro assembler which gives symbolic access to 8051 hardware features, the RL51 linker and relocator program that links modules generated by ASM-51, CONV51 which enables software written for the MCS-48 family to be up-graded to run on the 8051, and the LIB51 Librarian which programmers can use to create and maintain libraries of software object modules. Use with the DOS operating system (version 3.0 or later).
D86PLM51	PL/M-51 Software Package (DOS version) — Consists of the PL/M-51 compiler that provides high-level programming language support, the LIB51 utility that creates and maintains libraries of software object modules, and the RL51 linker and relocator program that links modules generated by ASM-51 and PL/M-51 and locates the linked object modules to absolute memory locations. Use with the DOS operating system (version 3.0 or later).
I86ASM51	8051 Software Development Package (ISIS version) — Same as the D86ASM51 package except this one is for use with the Series III and Series IV.
I86PLM51	PL/M-51 Software Package — Same as the D86PLM51 package except this one is for use with the Series III and Series IV.
D86EDIEU	AEDIT text editor for use with the DOS operating system.

ICE™-5100/452 IN-CIRCUIT EMULATOR FOR THE UPI™-452 FAMILY OF PROGRAMMABLE I/O PROCESSORS

- **Precise, full-speed, real-time emulation of the UPI™-452 family of I/O processors**
- **64 KB of mappable high-speed emulation memory**
- **254 24-bit frames of trace memory (16 bits trace program execution addresses and 8 bits trace external events)**
- **Serial link to the IBM* PC AT, PC XT (and DOS compatibles), and the Intellec® Series III/IV**
- **ASM-51 and PL/M-51 language support**
- **Full emulation and debug support for the FIFO buffer**
- **Built-in CRT-oriented text editor**
- **Symbolic debugging enables access to memory locations and program variables**
- **Four address breakpoints with in-range, out-of-range, and page breaks**
- **Equipped with the Integrated Command Directory (ICD™) that provides:**
  - **On-line help**
  - **Syntax guidance and checking**
  - **Dynamic command entry**
  - **Error checking**
  - **Command recall**
- **On-line disassembler and single-line assembler to help with code patching**

The ICE™-5100/452 in-circuit emulator is a high-level, interactive debugger that is used to develop and test the hardware and software of a target system based on the UPI™-452 family of I/O processors. The ICE-5100/452 emulator can be serially linked to an Intellec® Series III/IV or an IBM PC AT or PC XT. The emulator can communicate with the host system at standard baud rates up to 19.2K.



*IBM is a registered trademark of International Business Machines Corporation.

October 1986
Order Number: 280338

PRODUCT OVERVIEW

The ICE-5100/452 emulator provides full emulation support for the UPI-452 family of I/O processors. The UPI-452 family consists of the 83452, 87452, and the 80452.

The ICE-5100/452 emulator enables hardware and software development to proceed simultaneously. With the ICE-5100/452 emulator, prototype hardware can be added to the system as it is designed and software can be developed prior to the completion of the hardware prototype. Software and hardware integration can occur while the product is being developed.

The ICE-5100/452 emulator assists four stages of development:

- Software debugging
- Hardware debugging
- System integration
- System test

SOFTWARE DEBUGGING

The ICE-5100/452 emulator can be operated without being connected to the target system and before any of the user's hardware is available (provided external data RAM is not needed). In this stand-alone mode, the ICE-5100/452 emulator can be used to facilitate program development.

HARDWARE DEBUGGING

The ICE-5100/452 emulator's AC/DC parametric characteristics match the microcontroller's. The emulator's full-speed operation makes it a valuable tool for debugging hardware, including time-critical serial port, timer, and external interrupt interfaces.

SYSTEM INTEGRATION

Integration of software and hardware can begin when the emulator is plugged into the microcontroller socket of the prototype system hardware. Hardware can be added, modified, and tested immediately. As each section of the user's hardware is completed, it can be added to the prototype. Thus, the hardware and software can be system tested in real-time operation as each section becomes available.

SYSTEM TEST

When the prototype is complete, it is tested with the final version of the system software. The ICE-5100/452 emulator is then used for real-time emulation of the microcontroller to debug the system as a completed unit.

The final product verification test can be performed using the ROM orf EPROM version of the microcontroller. Thus, the ICE-5100/452 emulator provides the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

PHYSICAL DESCRIPTION

The ICE-5100/452 emulator consists of the following components (see Figure 1):

- Power supply
- AC and DC power cables
- Controller pod
- Serial cable (host-specific)
- User probe assembly (consisting of the processor module and the user cable)
- Crystal power accessory (CPA)
- 68-pin PGA target adaptor
- Clips assembly
- Software (includes the ICE-5100/452 emulator software, diagnostic software, and a tutorial)

The controller pod contains 64 KB of emulation memory, 254- by 24-bit frames of trace memory, and the control processor. In addition, the controller pod houses a BNC connector that can be used to connect up to 10 multi-ICE compatible emulators for synchronous starting and stopping of emulation.

The serial cable connects the host system to the controller pod. The serial cable supports a subset of the RS-232C signals.

The user probe assembly consists of a user cable and a processor module. The processor module houses the emulation processor and the interface logic. The target adaptor connects to the processor module and provides an electrical and mechanical interface to the target microcontroller socket.

The crystal power accessory (CPA) is a small, detachable board that connects to the controller pod and enables the ICE-5100/452 emulator to run in stand-alone mode. The target adaptor plugs into the socket on the CPA; the CPA then supplies clock and power to the user probe.

The clips assembly enables the user to trace external events. Eight bits of data are gathered on the rising edge of $\overline{\text{PSEN}}$ during opcode fetches. The clips information can be displayed using the CLIPS option with the PRINT command. Trace qualification input and output lines are also provided on the clips pod for connection to test equipment.

The ICE-5100/452 emulator software supports mnemonics, object file formats, and symbolic references generated by Intel's ASM-51 and PL/M-51 programming languages. Along with the ICE-5100/452 emulator software is a customer confidence test disk with diagnostic routines that check the operation of the hardware.

The on-line tutorial is written in the ICE-5100 command language. Thus, the user is able to interact with and use the ICE-5100/452 emulator while executing the tutorial.

A comprehensive set of documentation is provided with the ICE-5100/452 emulator.

ICE™-5100/452 EMULATOR FEATURES

The ICE-5100/452 emulator has been created to assist a product designer in developing, debugging and testing designs incorporating the UPI-452 family of I/O processors. The following sections detail some of the ICE-5100/452 emulator features.

EMULATION

Emulation is the controlled execution of the user's software in the target hardware or in an artificial hardware environment that duplicates the microcontroller of the target system. Emulation is a transparent process that happens in real-time. The execution of the user software is facilitated with the ICE-5100/452 command language.

MEMORY MAPPING

The memory space for the 452 microcontroller and its target hardware is fully accessible through the emulator. The ICE-5100/452 emulator refers to four physically distinct memory spaces, as follows:

- CODE — references program memory
- IDATA — references internal data memory
- RDATA — references special function register memory
- XDATA — references external data memory

ICE-5100/452 emulator commands that access memory use one of the special prefixes (e.g., CODE) to specify the memory space.

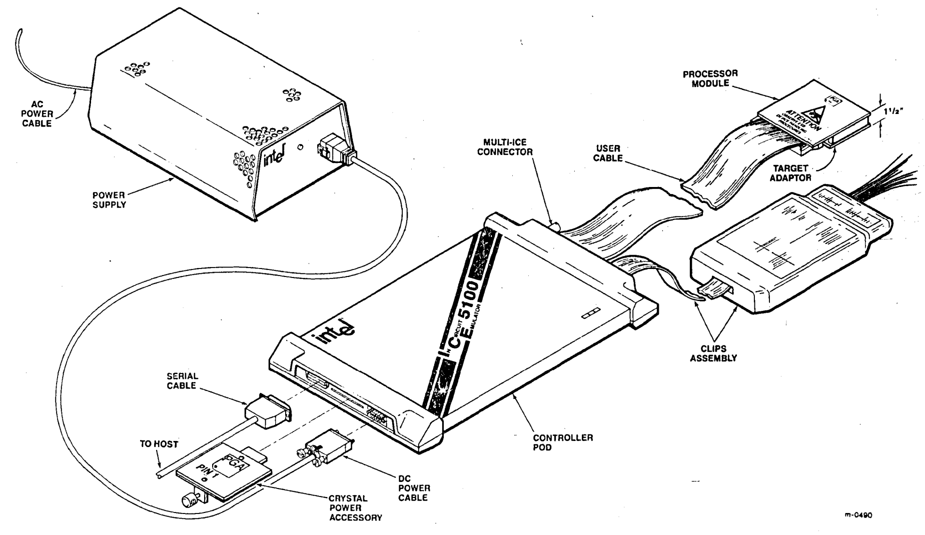


Figure 1. The ICE™-5100/452 Emulator Hardware

The ICE-5100/452 emulator has the following FIFO buffer access commands:

• FCLR	Resets the entire FIFO buffer, or resets either the input portion or the output portion.
• FDUMP	Performs a non-destructive read of the input or output FIFO buffer.
• FWRITE	Loads values into the input or output FIFO buffer.
• FREAD	Simulates a component read or a host system read of the FIFO buffer.
• FREEZE	Enables or disables the FIFO buffer access to the host CPU.

The microcontroller's special function registers and register bits can be accessed mnemonically (e.g., DPL, TCON, CY) with the ICE-5100/452 emulator software.

Data can be displayed or modified in one of three bases: hexadecimal, decimal, and binary. Data can also be displayed or modified in one of two formats: ASCII and unsigned integer. Program code can be disassembled and displayed as ASM-51 assembler mnemonics. Code can be modified with standard ASM-51 statements using the built-in single-line assembler.

Symbolic references can be used to specify memory locations. A symbolic reference is a procedure name, line number, program variable, or label in the user program that corresponds to a location.

Some typical symbolic functions include:

- Changing or inspecting the value of a program variable by using its symbolic name to access the memory location.
- Defining break and trace events using symbolic references.
- Referencing variables as primitive data types. The primitive data types are ADDRESS, BIT, BOOLEAN, BYTE, CHAR (character), and WORD.

The ICE-5100/452 emulator maintains a virtual symbol table (VST) for program symbols. A maximum of 61 KB of host memory space is available for the VST. If the VST is larger than 61 KB, the excess is stored on available host system disk space and is paged in and out as needed. The size of the VST is limited only by the disk capacity of the host system.

BREAKPOINT SPECIFICATIONS

Breakpoints are used to halt a user program in order to examine the effect of the program's execution on the target system. The ICE-5100/452 emulator supports three different types of break specifications:

- Specific address break — a single address point can be specified to halt emulation.
- Range break — an arbitrary range of addresses can be specified to halt emulation. Program execution within or, optionally, outside the range halts emulation.
- Page break — up to 256 page breaks can be specified to halt emulation. A page break is defined as a range of addresses that is 256-bytes long and begins on a 256-byte address boundary.

Break registers are user-defined debug definitions used to create and store breakpoint definitions. Break registers can contain multiple breakpoint definitions and can optionally call debug procedures when emulation halts.

TRACE SPECIFICATIONS

Tracing can be triggered using specifications similar to those used for breaking. Normally, the ICE-5100/452 emulator traces program activity while the user program is executing. With a trace specification, tracing can be triggered to occur only when specific conditions are met during execution. Up to 254 24-bit frames of trace information are collected in a buffer during emulation. Sixteen of the 24 bits trace instruction execution addresses, and 8 bits capture external events (CLIPS).

The trace buffer display is similar to an ASM-51 program listing as shown in Figure 2. The PRINT command enables the user to selectively display the contents of the trace buffer. The user has the option of displaying the clips information as well as disassembled instructions.

PROCEDURES

Debugging procedures (PROCs) are a user-named group of ICE-5100/452 commands that are executed as one command. PROCs enable the user to define several commands in a named block structure. The commands are executed by entering the name of the PROC. The PROC bodies are a simple DO. . . END construct.

PROCs can simulate missing hardware or software, collect debug information, and execute high-level software patches. PROCs can be copied to text files on disk, then recalled for use in later test sessions. PROCs can also serve as program diagnostics, implementing ICE-5100/452 emulator commands or user-defined definitions for special purposes. PROCs can also be used to set breakpoints.

ON-LINE SYNTAX MENU

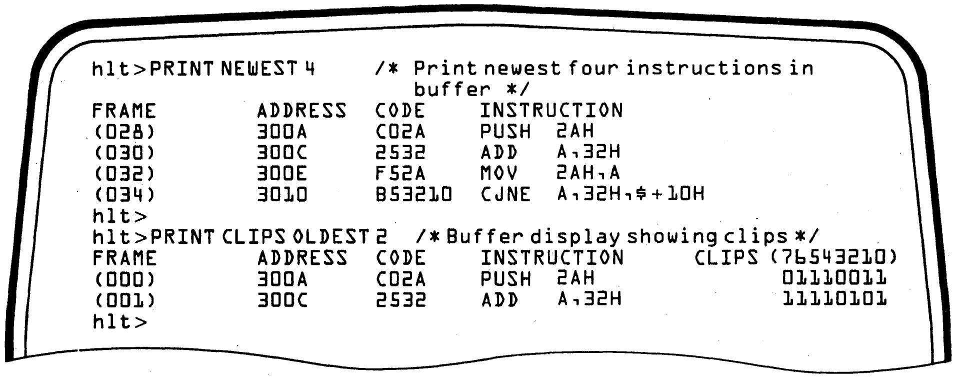
A special syntax menu, called the Integrated Command Directory (ICD), similar to the one used for the I²ICE™ system and the VLSiCE-96 emulator, aids in creating syntactically correct command lines. Figure 3 shows an example of the ICD and how it changes to reflect the options available for the GO command.

HELP

The HELP command provides ICE-5100/452 emulation command assistance via the host system terminal. On-line HELP is available for the ICE-5100/452 emulator commands shown in Figure 4.

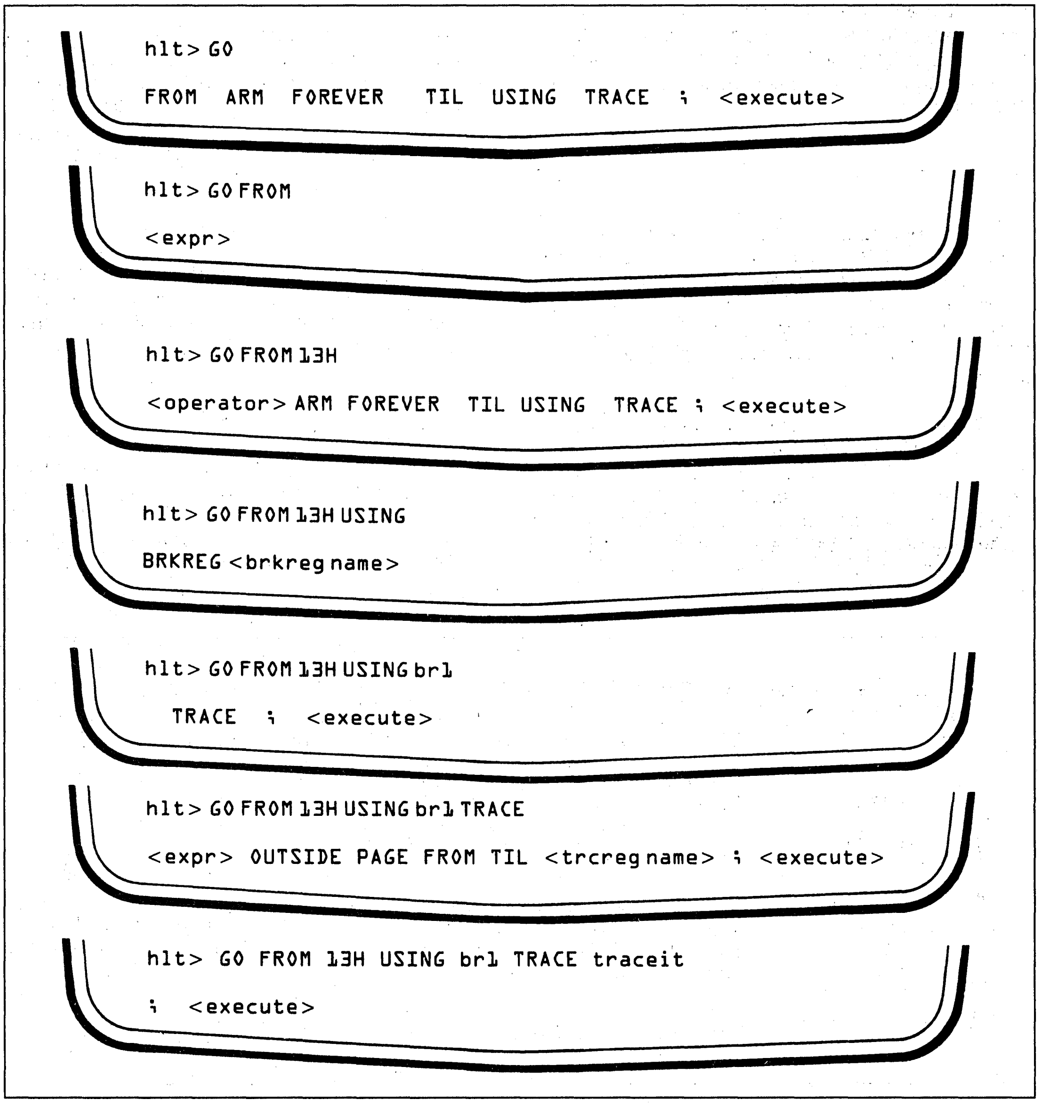
DESIGN CONSIDERATIONS

The height of the processor module and the target adaptor need to be considered for target systems. Allow at least 1½ inches (3.8 cm) of space to fit the processor module and target adaptor. Figure 5 shows the dimensions of the processor module.



```
hlt>PRINT NEWEST 4      /* Print newest four instructions in
                           buffer */
FRAME        ADDRESS  CODE    INSTRUCTION
(028)        300A     C02A    PUSH  2AH
(030)        300C     2532    ADD   A,32H
(032)        300E     F52A    MOV   2AH,A
(034)        3010     B53210  CJNE  A,32H,$+10H
hlt>
hlt>PRINT CLIPS OLDEST 2  /* Buffer display showing clips */
FRAME        ADDRESS  CODE    INSTRUCTION       CLIPS (76543210)
(000)        300A     C02A    PUSH  2AH                 01110011
(001)        300C     2532    ADD   A,32H               11110101
hlt>
```

Figure 2. Selected Trace Buffer Displays



```
hlt> GO
FROM  ARM  FOREVER  TIL  USING  TRACE  ;  <execute>
```

```
hlt> GO FROM
<expr>
```

```
hlt> GO FROM 13H
<operator> ARM FOREVER  TIL USING  TRACE ; <execute>
```

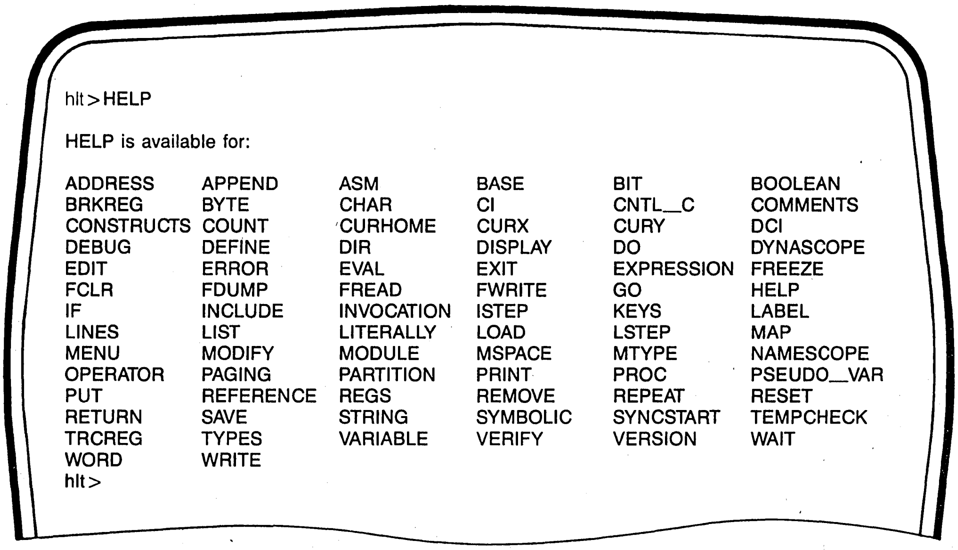
```
hlt> GO FROM 13H USING
BRKREG <brkreg name>
```

```
hlt> GO FROM 13H USING br1
  TRACE  ;  <execute>
```

```
hlt> GO FROM 13H USING br1 TRACE
<expr> OUTSIDE PAGE FROM TIL <trcreg name> ; <execute>
```

```
hlt> GO FROM 13H USING br1 TRACE traceit
;  <execute>
```

Figure 3. The Integrated Command Directory for the GO Command



```
hlt>HELP

HELP is available for:

ADDRESS     APPEND      ASM         BASE        BIT         BOOLEAN
BRKREG      BYTE        CHAR        CI          CNTL__C     COMMENTS
CONSTRUCTS  COUNT       CURHOME     CURX        CURY        DCI
DEBUG       DEFINE      DIR         DISPLAY     DO          DYNASCOPE
EDIT        ERROR       EVAL        EXIT        EXPRESSION  FREEZE
FCLR        FDUMP       FREAD       FWRITE      GO          HELP
IF          INCLUDE     INVOCATION  ISTEP       KEYS        LABEL
LINES       LIST        LITERALLY   LOAD        LSTEP       MAP
MENU        MODIFY      MODULE      MSPACE      MTYPE       NAMESCOPE
OPERATOR    PAGING      PARTITION   PRINT       PROC        PSEUDO__VAR
PUT         REFERENCE   REGS        REMOVE      REPEAT      RESET
RETURN      SAVE        STRING      SYMBOLIC    SYNCSTART   TEMPCHECK
TRCREG      TYPES       VARIABLE    VERIFY      VERSION     WAIT
WORD        WRITE
hlt>
```

Figure 4. HELP Menu

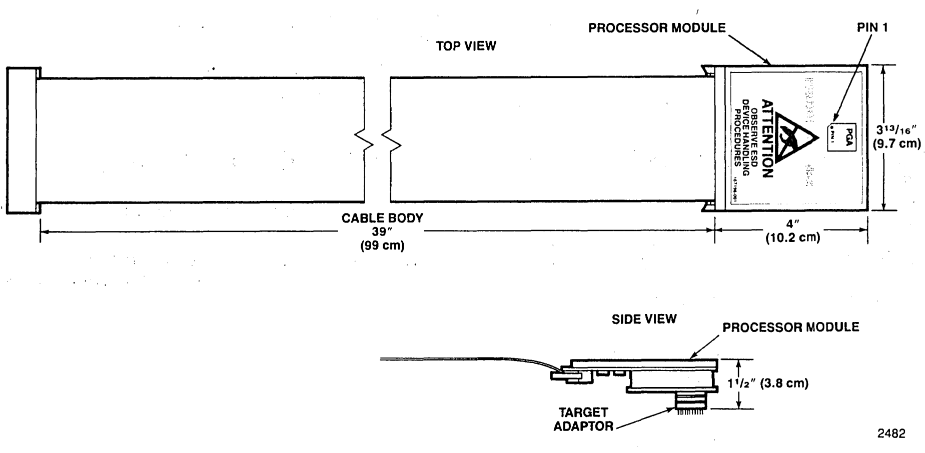


Figure 5. Processor Module Dimensions

ELECTRICAL CONSIDERATIONS

The emulation processor's user-pin timings and loadings are identical to the 452 component, except as follows:

- Up to 25 pf of additional pin capacitance is contributed by the processor module and target adaptor assemblies.

HOST REQUIREMENTS

- IBM PC AT or PC XT (or DOS compatible) with 512 KB of available RAM and a hard disk running under the DOS 3.0 (or later) operating system.
- Intellec Series III/IV microcomputer development system running the ISIS or iNDX operating system respectively, with at least 512 KB of application memory resident.
- Disk drives — dual floppy or one hard disk and one floppy drive required.

ICE™-452 EMULATOR SOFTWARE PACKAGE

- ICE-5100/452 emulator software
- ICE-5100/452 confidence tests
- ICE-5100 tutorial software

EMULATOR PERFORMANCE

Memory

Mappable full-speed emulation code memory	64 KB	Mappable to user or ICE-5100/452 emulator memory in 4 KB blocks on 4 KB boundaries.
Trace memory		254 × 24 bit frames.
Virtual Symbol Table		A maximum of 61 KB of host memory space is available for the virtual symbol table (VST). The rest of the VST resides on disk and is paged in and out as neded.

PHYSICAL CHARACTERISTICS

CONTROLLER POD

Width	8¼″	(21 cm)
Height	1½″	(3.8 cm)
Depth	13½″	(34.3 cm)
Weight	4 lbs	(1.85 kg)

USER CABLE

The user cable is 3 feet (approximately 1 m).

PROCESSOR MODULE

(With the target adaptor attached.)

Width	3-13/16″	(9.7 cm)
Height	4″	(10.2 cm)
Depth	1½″	(3.8 cm)

POWER SUPPLY

Width	7-5/8″	(18.1 cm)
Height	4″	(10.06 cm)
Depth	11″	(27.97 cm)
Weight	15 lbs	(6.1 kg)

SERIAL CABLE

The serial cable is 12 feet (3.6 m).

ELECTRICAL CHARACTERISTICS

POWER SUPPLY

100-120 V or 200-240 V (selectable)
50-60 Hz
2 amps (AC max) 9 120 V
1 amp (AC max) 9 240 V

ENVIRONMENTAL CHARACTERISTICS

Operating temperature	+10°C to +40°C (50°F to 104°F)
Operating humidity	Maximum of 85% relative humidity, non-condensing

ORDERING INFORMATION

EMULATOR HARDWARE AND SOFTWARE

Order Code	Description
1452KITAD	This kit contains the ICE-5100/452 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with an IBM PC AT or PC XT. The kit also includes the 8051 Software Development Package and the AEDIT text editor for use on DOS systems. (Requires software license.)
1452KITD	This kit is the same as the 1452KITAD kit excluding the 8051 Software Development Package and the AEDIT text editor. (Requires software license.)
1452KITAS	This kit contains the ICE-5100/452 user probe assembly, power supply and cables, serial cables, target adaptor, CPA, ICE-5100 controller pod, software, and documentation for use with Intel hosts (Series III/IV). The kit also includes the 8051 Software Development Package and the AEDIT text editor for use on the Series III/IV. (Requires software license.)
I452KITS	This kit is the same as the I452KITAS kit excluding the 8051 Software Development Package and the AEDIT text editor. (Requires software license.)

SOFTWARE ONLY

Order Code	Description
SA452D	This kit contains the host, probe, diagnostic, and tutorial software on 5¼″ disks for use on an IBM PC AT or PC XT (requires DOS 3.0 or later). (Requires software license.)
SA452S	This kit contains the host, probe, diagnostic and tutorial software on 8″ disks (both single-density and double-density) for use on a Series III, and on 5¼″ disks for use on a Series IV. (Requires software license.)

Other Useful Debug and Development Support Products

Order Code	Description
D86ASM51	**8051 Software Development Package** (DOS version) — Consists of the ASM-51 macro assembler which gives symbolic access to 8051 hardware features; the RL51 linker and relocator program that links modules generated by ASM-51; CONV51 which enables software written for the MCS-48 family to be upgraded to run on the 8051, and the LIB51 Librarian which programmers can use to create and maintain libraries of software object modules. Use with the DOS operating system (version 3.0 or later).
D86PLM51	**PL/M-51 Software Package** (DOS version) — Consists of the PL/M compiler which provides high-level programming langujage support; the LIB51 utility that creates and maintains libraries of software object modules, and the RL51 linker and relocator program that links modules generated by ASM-51 and PL/M-51 and locates the linked object modules to absolute memory locations. Use with the DOS operating system (version 3.0 or later).
I86ASM51	**8051 Software Development Package** (ISIS version) — Same as the D86ASM51 package except this one is for use with the Series III.
I86PLM51	**PL/M Software Package** — Same as the D86PLM51 package except this one is for use with the Series III and the Series IV.
D86EDINL	AEDIT text editor for use with the DOS operating system.

intel®

APPLICATION NOTE

AP-239

November 1986

Customer Applications of the EMV-88 Emulations Vehicle

BILL ALLEN
DSO PRODUCT MARKETING
FRED MOSEDALE
DSO TECHNICAL PUBLICATIONS

Order Number: 280105-001

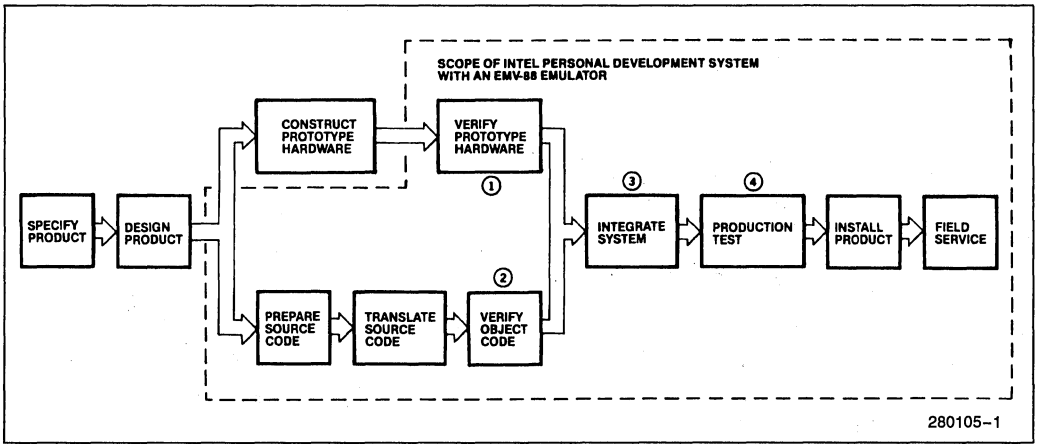


Figure 1. Typical Microcomputer Process

INTRODUCTION

Early customers' experiences with the EMV-88 emulation vehicle have shown the power, versatility, and benefits of this emulator. The EMV-88 emulator plugs into an Intel Personal Development system (iPDS™) and aids in the development and debugging of user-designed 8088 systems.

To aid new and potential users of the EMV-88, this application note summarizes applications and debugging procedures of several early users of the EMV-88 emulator.

THE MICROCOMPUTER DEVELOPMENT PROCESS

Designing a product that contains a microcomputer requires close coordination of two separate but highly dependent design efforts: hardware development and software development.

Hardware development involves planning the microprocessor chip's interaction with associated logic, memory, peripheral circuits, and specialized circuits.

Software development involves programming the microcomputer system to perform the required tasks. The resulting program eventually resides in the product's memory.

These two development efforts can be accomplished independently, but it is more efficient to work on them together. Successful designs make maximum use of the hand-in-hand nature of hardware and software. In addition, real-world designing is an iterative process. Each step in the design process may involve the debugging and re-design of previous steps. A change in hardware may involve a corresponding change in software and vice versa.

Figure 1 illustrates a typical microcomputer development process. The iPDS system and the EMV-88 emulator are two major design aids that Intel offers to hardware and software designers. Note the areas of the design process where a microcomputer development system and an emulator aid in the design of a product. (Numbers 1 through 4 shown in Figure 1 are used later in this application note.)

Features of the iPDS™ System and the EMV-88 Emulator

The iPDS system and EMV-88 emulator offer the following resources.

- A stand-alone computer with dual processors (optional), memory, mass storage, and a disk-based operating system.
- Development system software such as assemblers, high-level language compilers, and EMV-88 debugging software.
- An interface (using the EMV-88 cable) to the prototype hardware. This allows you to check each piece of your prototype hardware as it is developed.
- The EMV-88 mapping capability. This allows you to borrow the memory form the EMV-88 until prototype hardware is available.
- The EMV-88 break and trace ability. This allows you to specify the conditions under which emulation stops or tracing occurs.

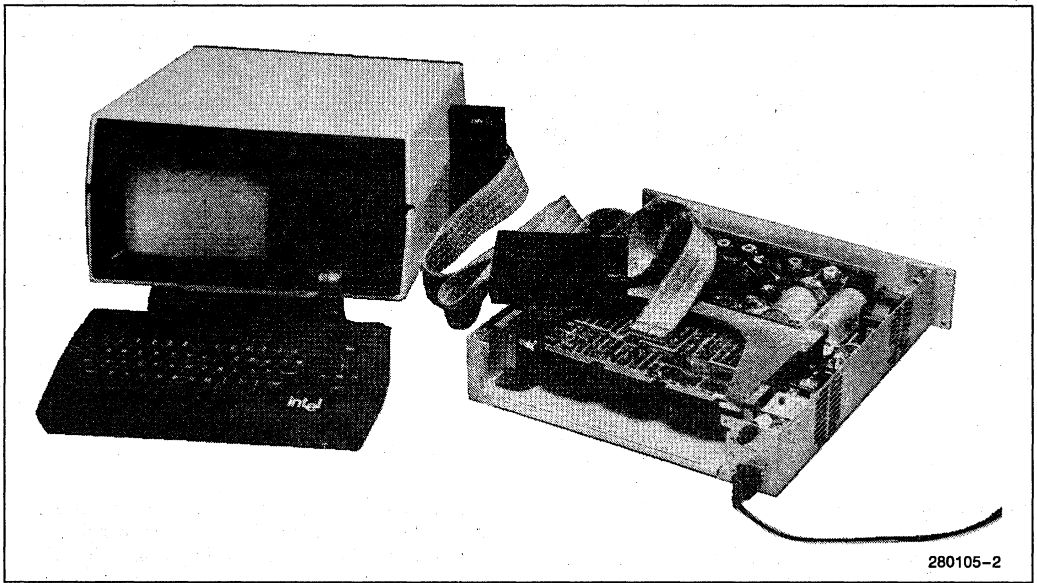


Figure 2. The iPDS™ System and the EMV-88 Emulator Connected to a User Prototype

- Availability of other plug-in modules for PROM programming and for emulation of other processors (8051, 8044, and 8085).

Figure 2 shows the EMV-88 emulator inserted in the side of the iPDS system. The EMV-88 emulator is shown connected to a user prototype board.

THE USERS

To obtain information for this application note, three early EMV-88 users were visited. Users were asked a variety of questions: How did the EMV-88 system save them time? What debugging procedures were especially useful? What EMV-88 features proved important in developing the users' designs? In retrospect, what features or techniques might have been used earlier to speed development and debugging even more?

The customers visited were the following:

— An established company originally specializing in water and waste water control devices. Now it is expanding to provide automatic control and monitoring devices. This company recently began designing microcontrollers and microprocessors into its products.

— An established communications and antenna company that was designing an 8088-based system to control the antenna for a satellite communications system.

— A new company that was designing a computerized system for monitoring automatic manufacturing process control machines.

All the customers had iPDS systems with EMV-88 plug-in emulators. (The customers also had other iPDS plug-in options, including PROM programming modules.)

USER APPLICATIONS AND DEBUGGING PROCEDURES

New and potential EMV-88 users will be interested in two kinds of information supplied by the early EMV-88 users:

(1) the different functions the EMV-88 emulators can be used to perform in early phases of a product's life cycle, and

(2) some specific EMV-88 debugging techniques that proved useful to the users. The following two main sections focus on these topics.

EMV-88 Emulator Functions in Early Phases of a Product's Life Cycle

As a product is developed, debugged, and released to customers, the EMV-88 emulator can accomplish a variety of tasks during early phases of the product's life cycle. In particular, the iPDS system with the EMV-88 emulator proved to be especially productive in completing the following tasks. (The task numbers are also shown in circles in Figure 1.)

Task 1	Verifying hardware
Task 2	Verifying software
Task 3	Integrating prototype hardware and software
Task 4	Production testing

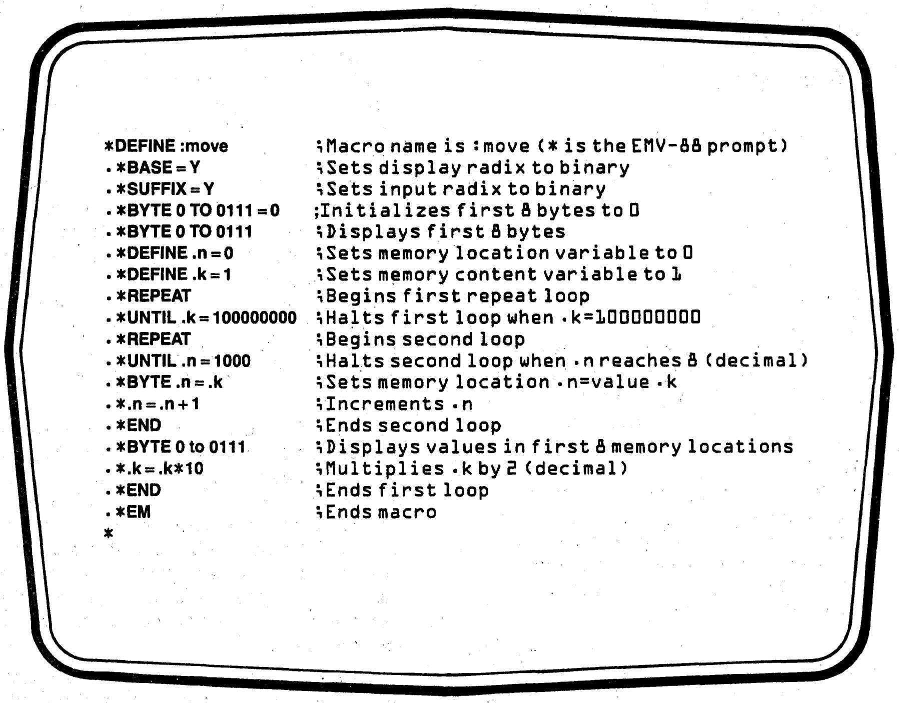
The EMV-88 system helped early users accomplish these tasks as described below:

- **Exercised hardware and software in real time** (tasks 1, 2, and 3): Without an emulator, users would only have been able to check their prototypes by loading their programs into EPROMs and running the programs. Then, when bugs were detected and corrected, the revised programs would have to be loaded into the EPROMs. This cycle would have to be repeated each time a bug was found. However, with the EMV-88 emulator, users did not need to load programs into EPROMs. The program resided in emulator memory and could easily be modified and retested. Thus, the emulator saved users time and provided flexibility in modifying programs.
- **For a prototype system with large programs, the EMV-88 emulator was used to supplement main development systems** (task 1): One user had very large programs under development. The user's large development systems were tied up with software development and could not be used with their emulators to test hardware. To speed up debugging, the iPDS system with its EMV-88 emulator was pressed into service. Short EMV-88 macros were written to exercise particular portions of the hardware. Thus, hardware development and testing could continue despite the unavailability of emulators on the large development systems.
- **Patched around missing sections of code to allow emulation** (tasks 2 and 3): Because the EMV-88 emulator allows users to patch code, whenever a section of code is incomplete or contains a bug, users can patch around it. One user was able to begin debugging the prototype before software development was completed. At the end of sections of completed code, EMV-88 commands were used to patch in a command to jump from the last line of code in one section to the first line of code in the next available section. Some of the activities of the still-to-be-completed code were also simulated with EMV-88 commands in the patch.
- **Resolved disputes about whether bugs were in hardware or software** (task 3): Because the iPDS system and its EMV-88 emulator can control and examine both user hardware and user software, it is relatively easy to determine whether a bug originates in software or hardware. For example, users took advantage of the EMV-88 emulator's single-stepping capability to determine at which line of code undesired values were generated. Then, carefully controlled emulation combined with the use of a logic analyzer allowed users to pinpoint the source of the problem. As a result, fingerpointing by software and hardware team members quickly came to an end.
- **Provided remote site testing of prototype hardware and software** (task 3): One user could not fully test the prototype hardware and software because the environment in which the prototype was intended to run could not be duplicated in the development area. Because the iPDS system and the EMV-88 are portable, they were easily moved to a site remote from the development area. Then, a full debugging session in the prototype's intended environment took place.
- **Tested early manufactured systems** (task 4): When the earliest boards have been manufactured, there must be a way to test them. If a complete board-testing system is not yet in place, the EMV-88 can act as a hardware tester. After test programs are written for the EMV-88, and the emulator is connected to a new board, users can quickly determine whether flaws exist in the manufacturing process. If tests are skillfully written, hardware areas that are failing can be pinpointed.
- **Used to troubleshoot early customer systems** (task 4): Despite careful quality control, not all bugs in the design and manufacturing processes may be detected. Early customers may report problems they are having with the product. If swapping hardware and/or software corrects the problem, the defective hardware and/or software can be returned for troubleshooting. The EMV-88 emulator can track down the problem. It is important to determine the source of the problem so that, if need be, design changes or manufacturing changes can quickly be initiated.

As is evident from the preceding list, early users found a variety of tasks for the EMV-88 emulator to perform during product development and manufacturing. One user noted that the iPDS system is an excellent development system for both young and mature companies. Its low price, versatility, and portability make the iPDS system with the EMV-88 emulator an investment that returns its cost many times.

Early EMV-88 Users' Debugging Techniques

Six debugging problems early users encountered have been selected to illustrate a variety of EMV-88 emulator's capabilities.



```
*DEFINE :move            ;Macro name is :move (* is the EMV-88 prompt)
.*BASE=Y                 ;Sets display radix to binary
.*SUFFIX=Y               ;Sets input radix to binary
.*BYTE 0 TO 0111=0       ;Initializes first 8 bytes to 0
.*BYTE 0 TO 0111         ;Displays first 8 bytes
.*DEFINE .n=0            ;Sets memory location variable to 0
.*DEFINE .k=1            ;Sets memory content variable to 1
.*REPEAT                 ;Begins first repeat loop
.*UNTIL .k=100000000     ;Halts first loop when .k=100000000
.*REPEAT                 ;Begins second loop
.*UNTIL .n=1000          ;Halts second loop when .n reaches 8 (decimal)
.*BYTE .n=.k             ;Sets memory location .n=value .k
.*.n=.n+1                ;Increments .n
.*END                    ;Ends second loop
.*BYTE 0 to 0111         ;Displays values in first 8 memory locations
.*.k=.k*10               ;Multiplies .k by 2 (decimal)
.*END                    ;Ends first loop
.*EM                     ;Ends macro
*
```

280105-3

Figure 3. Sample Macro for Testing Memory

PROBLEM 1: INITIAL CHECK OF PROTOTYPE HARDWARE (Task 1: Verify Hardware)

One early user made an initial check of prototype hardware with the EMV-88 emulator. Once the user's RAM, USART, and registers were in place in the hardware prototype, an initial hardware check was scheduled. Were the components installed and connected properly? (To ease hardware-software integration, efforts should first be made to isolate hardware defects independently of the prototype software.)

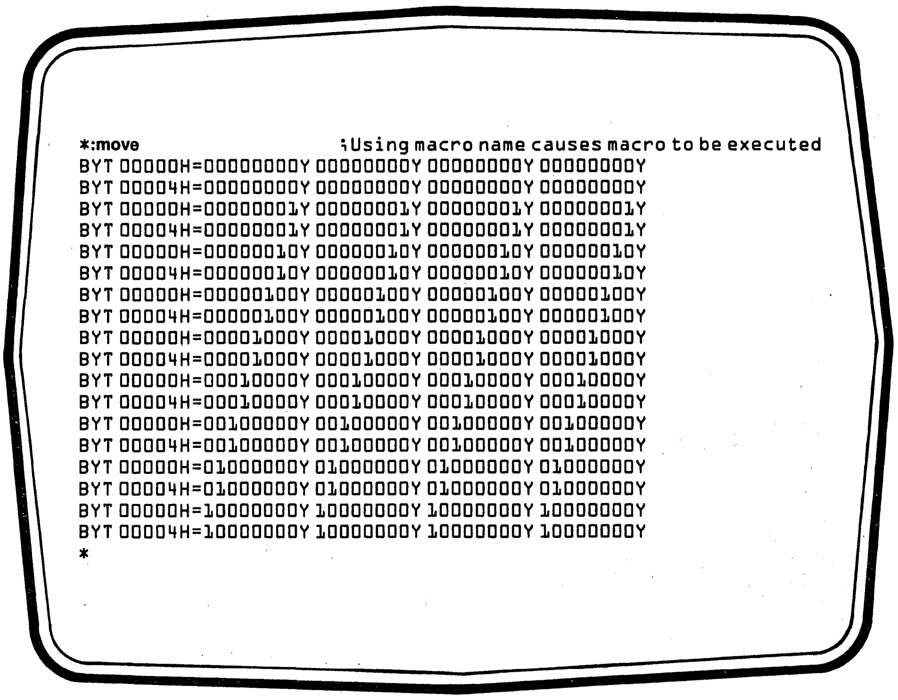
PROBLEM 1 SOLUTION

The EMV-88 user performed the following steps to check out prototype hardware.

1. Identified the addresses of all hardware elements to be tested.
2. Devised EMV-88 hardware test macros: Macros were created that wrote patterns of 1's and 0's to the memory devices and registers. The macros also were designed to read and display memory and register contents. (See Figures 3 and 4 for a sample macro that writes patterns of 1's and 0's to a small portion of memory. Also, see the appendix for information on EMV-88 commands.)
3. Executed the macros and observed the results on the IPDS system display screen.
4. Identified defective hardware areas: When an output value was different from an input value, the user executed memory interrogation commands (e.g., BYTE, WORD, DUMP) to confirm the location of defective hardware.

PROBLEM 2: WRONG INSTRUCTION EXECUTION SEQUENCE (Task 2: Verify Software)

When this user's prototype program was emulated, a portion of the program ran properly, but then it performed strangely—it "ran in the weeds." How can the EMV-88 emulator locate the area of program code where the execution sequence first begins to go wrong?



```
*:move                         ;Using macro name causes macro to be executed
BYT 00000H=00000000Y 00000000Y 00000000Y 00000000Y
BYT 00004H=00000000Y 00000000Y 00000000Y 00000000Y
BYT 00000H=00000001Y 00000001Y 00000001Y 00000001Y
BYT 00004H=00000001Y 00000001Y 00000001Y 00000001Y
BYT 00000H=00000010Y 00000010Y 00000010Y 00000010Y
BYT 00004H=00000010Y 00000010Y 00000010Y 00000010Y
BYT 00000H=00000100Y 00000100Y 00000100Y 00000100Y
BYT 00004H=00000100Y 00000100Y 00000100Y 00000100Y
BYT 00000H=00001000Y 00001000Y 00001000Y 00001000Y
BYT 00004H=00001000Y 00001000Y 00001000Y 00001000Y
BYT 00000H=00010000Y 00010000Y 00010000Y 00010000Y
BYT 00004H=00010000Y 00010000Y 00010000Y 00010000Y
BYT 00000H=00100000Y 00100000Y 00100000Y 00100000Y
BYT 00004H=00100000Y 00100000Y 00100000Y 00100000Y
BYT 00000H=01000000Y 01000000Y 01000000Y 01000000Y
BYT 00004H=01000000Y 01000000Y 01000000Y 01000000Y
BYT 00000H=10000000Y 10000000Y 10000000Y 10000000Y
BYT 00004H=10000000Y 10000000Y 10000000Y 10000000Y
*
```

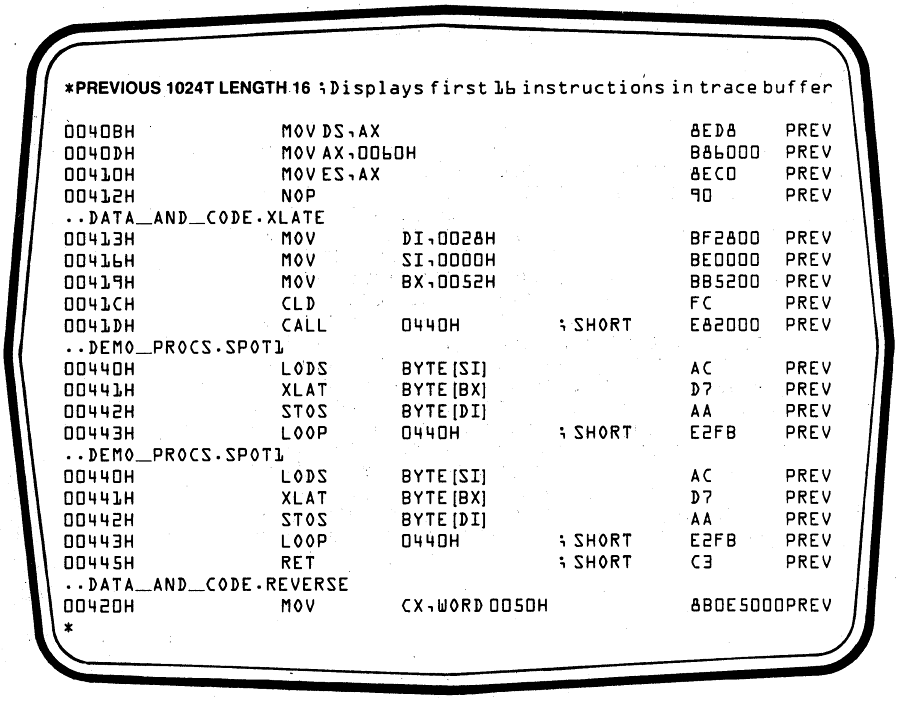
280105-4

Figure 4. Sample Display Resulting from Figure 3 Macro

PROBLEM 2 SOLUTION

The user performed the following steps to locate the area of code where code begins defective operation.

1. Emulated the program: The user executed the GO command with the FROM option; the program starting address was entered after FROM. (See the appendix for information on EMV-88 commands.)
2. Examined the trace buffer: The PREVIOUS command was used to scan through the 1K byte trace buffer. (See Figure 5 for a sample display using the PREVIOUS command. In Figure 5, the first 16 instructions in the 1K byte trace buffer are displayed.) The instructions stored at the very beginning of the buffer were incorrect. This implied that the problem was further back in program execution. The instruction address at the beginning of the trace buffer was noted.
3. Set a breakpoint: To make possible examination of the previous 1K bytes of the program execution sequence, the user set an execution breakpoint at the address identified in step 2.
4. Re-emulated. When emulation occurred using the new breakpoint, emulation halted at the point the previous trace buffer started collecting trace information. Now, the new trace buffer contained the preceding 1K bytes of executed instructions
5. Examined the trace buffer: Scanning through the new trace buffer contents the user came upon the program section where the execution sequence went awry. Study of the program section showed a programming error.
6. Patched code: Using the ORG (originate) and ASM (assemble) commands, the user created a patch. (See Figure 6 for a display of sample EMV-88 patching commands.) First the instruction pointer was moved to the location of the defective line of code using the ORG command. Then, the ASM command inserted a jump command to an unused area of memory. Using ORG and ASM, a patch of correct code was created at the unused memory location; the patch included a jump back to the instruction next after the line of defective code.



```
*PREVIOUS 1024T LENGTH 16 ;Displays first 16 instructions in trace buffer

0040BH          MOV DS,AX                                   8ED8      PREV
0040DH          MOV AX,0060H                                B86000    PREV
00410H          MOV ES,AX                                   8EC0      PREV
00412H          NOP                                         90        PREV
..DATA_AND_CODE.XLATE
00413H          MOV         DI,0028H                        BF2800    PREV
00416H          MOV         SI,0000H                        BE0000    PREV
00419H          MOV         BX,0052H                        BB5200    PREV
0041CH          CLD                                         FC        PREV
0041DH          CALL        0440H          ; SHORT          E82000    PREV
..DEMO_PROCS.SPOT1
00440H          LODS        BYTE [SI]                       AC        PREV
00441H          XLAT        BYTE [BX]                       D7        PREV
00442H          STOS        BYTE [DI]                       AA        PREV
00443H          LOOP        0440H          ; SHORT          E2FB      PREV
..DEMO_PROCS.SPOT1
00440H          LODS        BYTE [SI]                       AC        PREV
00441H          XLAT        BYTE [BX]                       D7        PREV
00442H          STOS        BYTE [DI]                       AA        PREV
00443H          LOOP        0440H          ; SHORT          E2FB      PREV
00445H          RET                        ; SHORT          C3        PREV
..DATA_AND_CODE.REVERSE
00420H          MOV         CX,WORD 0050H                   8B0E5000PREV
*
```

280105-5

Figure 5. Sample Display Using the PREVIOUS Command

7. Re-emulated: Emulation stopped at the breakpoint set in step 3.
8. Examined the trace buffer: This time the trace buffer showed that program execution followed the correct sequence. Thus, the patch fixed the problem.

PROBLEM 3: DEBUGGING IN A MULTI-TASKING ENVIRONMENT
(Task 2: Verify Software)

Programs that support multi-tasking can be difficult to debug when interrupts arrive that place the current task on the stack while another task is undertaken. One EMV-88 user performed the following steps to overcome this problem.

PROBLEM 3 SOLUTION

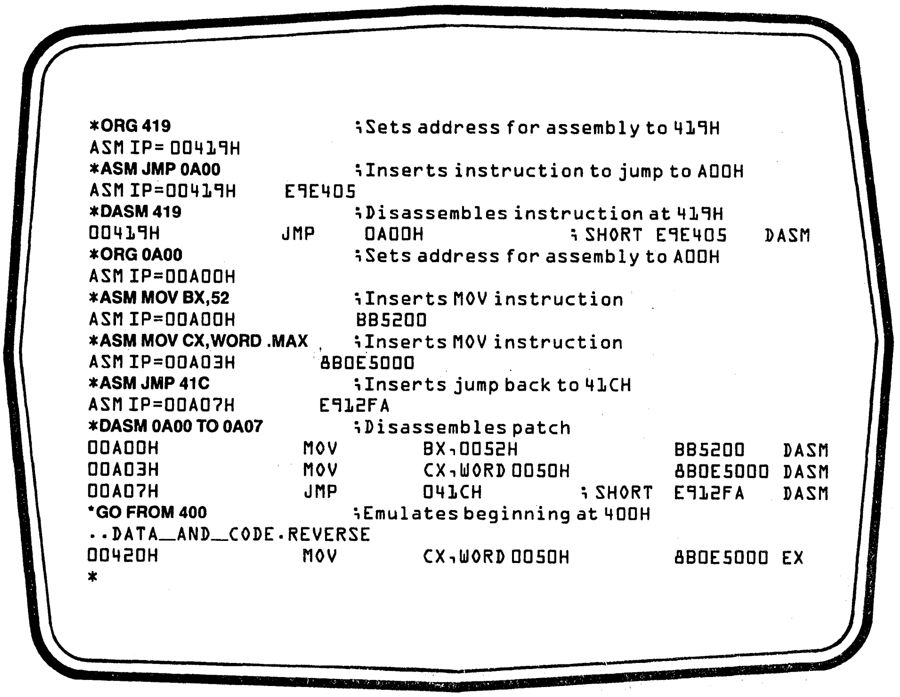
1. Cleared the interrupt enable flag: By entering IFF = 0, the 8088 interrupt enable flag was cleared. See Figure 7 for a display of register settings that shows the resulting IFF setting.
2. Emulated the code of interest: The user used the GO command with the FROM option to set a breakpoint and thus control emulation of the desired section of code. (See the appendix for information on EMV-88 commands.) With the interrupt enable flag cleared, trace information was collected without other tasks interrupting the trace data collection.
3. Re-enabled the interrupt enable flag.

PROBLEM 4: MEMORY NOT BEING ZEROED
(Task 3: Integrate Hardware and Software)

This user first employed the EMV-88 emulator to test a section of code that was supposed to zero memory. The test showed that memory was not being zeroed. What prevented the memory initialization?

PROBLEM 4 SOLUTION

Once it was clear that memory was not being zeroed, the user (whose iPDS system had the optional dual processors) followed these steps to identify what was preventing memory from being initialized.



```
*ORG 419                    ;Sets address for assembly to 419H
ASM IP= 00419H
*ASM JMP 0A00               ;Inserts instruction to jump to A00H
ASM IP=00419H      E9E405
*DASM 419                   ;Disassembles instruction at 419H
00419H             JMP    0A00H          ; SHORT E9E405     DASM
*ORG 0A00                   ;Sets address for assembly to A00H
ASM IP=00A00H
*ASM MOV BX,52              ;Inserts MOV instruction
ASM IP=00A00H               BB5200
*ASM MOV CX,WORD .MAX       ;Inserts MOV instruction
ASM IP=00A03H             8B0E5000
*ASM JMP 41C                ;Inserts jump back to 41CH
ASM IP=00A07H             E912FA
*DASM 0A00 TO 0A07          ;Disassembles patch
00A00H             MOV        BX,0052H                BB5200    DASM
00A03H             MOV        CX,WORD 0050H           8B0E5000  DASM
00A07H             JMP        041CH         ; SHORT   E912FA    DASM
*GO FROM 400                ;Emulates beginning at 400H
..DATA_AND_CODE.REVERSE
00420H             MOV        CX,WORD 0050H           8B0E5000  EX
*
```

280105-6

Figure 6. Sample Patch Commands

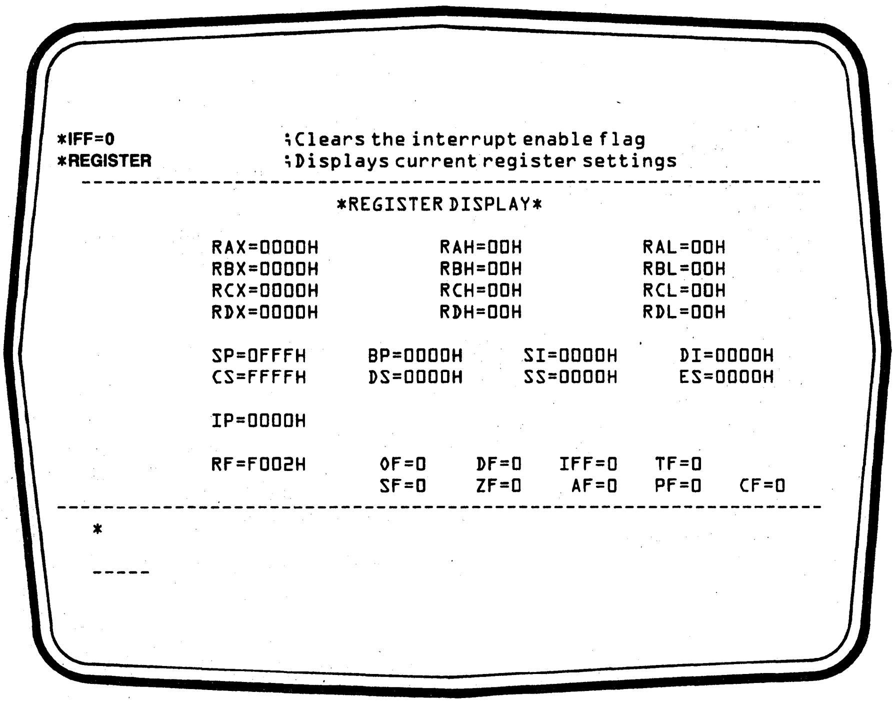
1. Used the B processor to locate code: The iPDS file display command (@) was used to scroll through the code listing to locate the line of code where memory zeroing began. The line that completed the zeroing operation was also located.
2. Set up a trace point and a breakpoint: After switching to the A processor, the user set a trace point and a breakpoint that began trace at the first line identified in step 1 and caused emulation to break at the other line identified in step 1. (See the appendix for information on EMV-88 commands.)
3. Initiated emulation using the GO command.
4. Examined the trace buffer: The trace buffer showed tha the expected data (FC) was read from program memory.
5. Connected a logic analyzer to an EMV-88 controller test signal: A logic analyzer was connected to the BRK test signal available on the EMV-88 controller module; the signal is useful in triggering a logic analyzer to capture data on the bus.
6. Re-emulated.
7. Examined the logic analyzer display: The logic analyzer display showed that FF was being received by the processor even though FC was being sent.
8. Connected an oscilloscope to the bus: The oscilloscope showed ringing on the bus. (The ringing was traced to a faulty extender card.) The ringing caused bus signals to be near threshold values. Low signals could be interpreted by the processor as high or low. Thus FC (1111 1100) could be interpreted as FF (1111 1111).

PROBLEM 5: DISPLAY UPDATE SIGNAL IS SLOW
(Task 3: Integrate Hardware and Software)

This user developed a system with a display that must be updated every second. However, in each five minute period, the display was updated one time less than it should be. The user needed to determine whether a counter implemented in software was not generating the correct update signal or whether the output from a separate timer board (that incremented the counter) was too slow.

PROBLEM 5 SOLUTION

This is a problem that became evident to the product development team after software and hardware were



```
*IFF=0                  ;Clears the interrupt enable flag
*REGISTER               ;Displays current register settings
-----------------------------------------------------------------
                       *REGISTER DISPLAY*

             RAX=0000H            RAH=00H            RAL=00H
             RBX=0000H            RBH=00H            RBL=00H
             RCX=0000H            RCH=00H            RCL=00H
             RDX=0000H            RDH=00H            RDL=00H

             SP=0FFFH      BP=0000H      SI=0000H      DI=0000H
             CS=FFFFH      DS=0000H      SS=0000H      ES=0000H

             IP=0000H

             RF=F002H       OF=0     DF=0    IFF=0    TF=0
                            SF=0     ZF=0     AF=0    PF=0    CF=0
-----------------------------------------------------------------
  *
  -----
```

280105–7

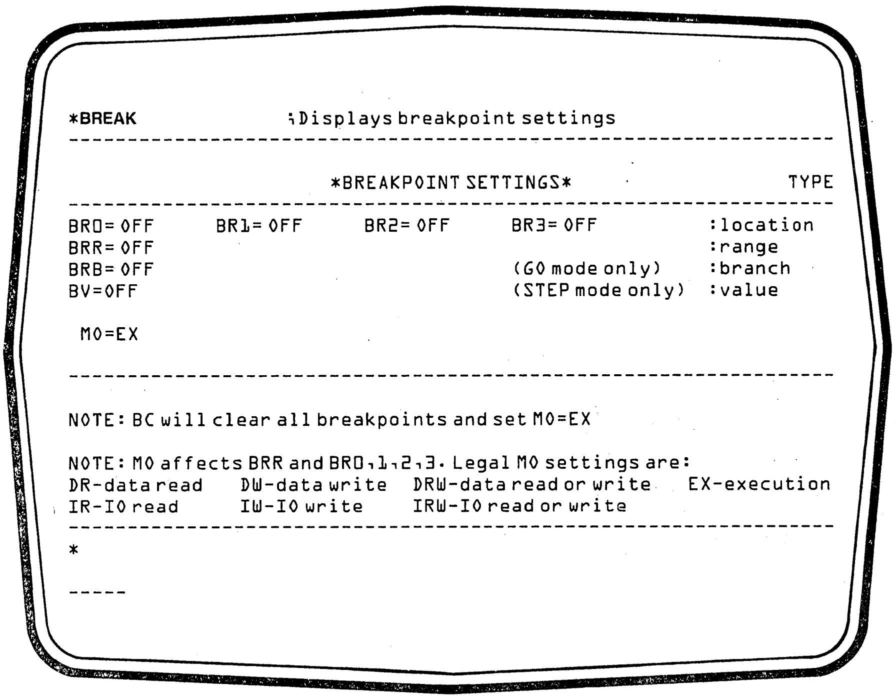
Figure 7. Sample REGISTER Display that Shows the New IFF Setting

integrated. It was unclear whether it was a software or hardware defect. The team employed the following steps with the EMV-88 emulator to locate the source of the problem.

1. Created a counter macro: A macro was created that counted each time the external board sent a signal to a specific input port of the 8088. (See the appendix for information on EMV-88 commands.) The macro also sent a signal to an ouput port when the counter reached the correct count. The team reasoned that if the problem still existed when they used the macro counter, the counter in the prototype software could be eliminated as the source of the problem.
2. Executed the macro and checked the output signal: The output port signal interval was slightly longer than the desired one second interval. Thus, the problem must be caused by the signal input to the counter.
3. Measured the input signal: The input signal was expected to occur at 0.500 second intervals. However, measurements showed that it occurred at longer intervals. It seemed, then, that the count board was to blame. However, examination of the board's specifications showed that the output of the board was ambiguously specified. In one place it gave the timer output as occurring at 0.500 intervals and in other places the interval was specified with a +0.016 second tolerance. So the cause of the slow display update was neither a hardware defect nor a software defect. Rather, to blame were an ambiguous specification and the failure of the designers to look for and to take into account the tolerance of the timer's interval.

PROBLEM 6: READ-ONLY MEMORY IS WRITTEN TO
(Task 3: Integrate Hardware and Software)

One user encountered a situation in which a read-only area of memory was written to during program execution. The EMV-88 user performed the following steps to isolate the error.



```
*BREAK              ;Displays breakpoint settings
-----------------------------------------------------------------

                    *BREAKPOINT SETTINGS*                   TYPE
-----------------------------------------------------------------
BR0= OFF    BR1= OFF    BR2= OFF    BR3= OFF           :location
BRR= OFF                                               :range
BRB= OFF                            (GO mode only)     :branch
BV=OFF                              (STEP mode only)   :value

 MO=EX

-----------------------------------------------------------------

NOTE: BC will clear all breakpoints and set MO=EX

NOTE: MO affects BRR and BR0,1,2,3. Legal MO settings are:
DR-data read    DW-data write    DRW-data read or write    EX-execution
IR-IO read      IW-IO write      IRW-IO read or write
-----------------------------------------------------------------
*

-----
```

280105-8

Figure 8. BREAK Display

PROBLEM 6 SOLUTION

1. Set a range breakpoint: By using the BREAK command (or FUNCTION-2), the current breakpoint settings were displayed. (See Figure 8 for a display of EMV-88 breakpoint settings. Also, see the appendix for information on EMV-88 commands.) The breakpoint mode was set to data write (MO = DW), and the range breakpoint was set to the memory range of interest. As a result of these settings, emulation breaks if a data write occurs within the specified memory range.
2. Emulated.
3. Examined the trace buffer: Examined the previous 16 instructions in the trace buffer (by entering PREVIOUS 16). Defective code was discovered.
4. Patched and tested the code. See Problem 2 for an account of patching procedures.

SUMMARY

Early users of the EMV-88 emulator used the emulator to perform the following functions in the early stages of their products; life cycles.

- Exercised hardware and software in real time.

- For a prototype system with large programs, the EMV-88 emulator was used to supplement main development systems.
- Patched around missing sections of code to allow emulation when some portions of code were unavailable.
- Resolved disputes about whether bugs were in hardware or software.
- Tested prototype hardware and software at a remote site.
- Tested early manufactured systems.
- Helped in troubleshooting early customer system returns.

In addition, early users showed that the resources of the EMV-88 software and hardware can be used to cope with a wide variety of debugging problems. The EMV-88 emulator performed the following tasks:

- Made an initial check of prototype hardware.
- Located code that caused the instruction execution sequence to be wrong.
- Devised a way to debug in a multi-tasking environment.
- Identified the reason that memory was not being zeroed.
- Isolated the cause for a counter counting too slowly.
- Located code that was permitting writing to read-only memory.

Finally, early EMV-88 customers also made use of other iPDS plug-in modules. They used other emulation vehicles to debug other portions of their hardware and software that were designed around other Intel processors; they also used PROM programming modules to load their debugged code into their prototype system PROMs.

New users of the iPDS system and EMV-88 emulator are encouraged to make full use of these systems' capabilities and resources to perfect their products. Only some of the capabilities of the EMV-88 emulator and the iPDS system have been described here. Review the iPDS system and EMV-88 emulator manuals to gain full knowledge of the command sets and options.

APPENDIX: SUMMARY OF EMV-88 COMMANDS AND COMMAND CATEGORIES

APPENDIX: SUMMARY OF EMV-88 COMMANDS AND COMMAND CATEGORIES

The EMV-88 emulator is a full symbolic emulator, and hence all commands and displays can be entered symbolically. The EMV-88 emulator and the user can thus communicate by referring to symbols defined in the user's source program or symbols defined during the debugging session. Ths following sections describe the command categories and Table 1 summarizes the EMV-88 commands.

UTILITY COMMANDS

Utility commands performs functions not directly related to the task of emulation and debugging. These commands gain access to the iPDS system resources and display information about the emulator.

DISPLAY/MODIFY COMMANDS

These commands change or display any register, port, or memory addressable by the iAPX-88 target system. They provide access to specific areas of the processor or target system and thus minimize extraneous display information.

EMULATION COMMANDS

Commands that control program execution or initiate emulation fall into this category. The GO, BREAK, and TRACE commands are in this category.

ADVANCED COMMANDS

The advanced commands offer an easy way to increase the debugging capability of this product. These advanced features allow the EMV-88 emulator command sequences to be combined, executed, and stored.

Table 1. Summary of EMV-88 Commands

Command Category	Command	Command Definition
Utility Commands	DEFINE	Defines symbol or macro.
	DOMAIN	Establishes default module.
	ENABLE/DISABLE	Controls expanded display.
	EVALUATE	Evaluates any expression.
	EXIT	Terminates EMV-88 session.
	HELP	Displays command syntax.
	INCLUDE	Loads a macro definition or a command file.
	LINE	Displays statement numbers and associated absolute addresses.
	LIST	Generates copy of emulation work session.
	LOAD	Loads object file in mapped memory.
	MODULE	Displays module names in EMV-88 module table.
	REMOVE	Deletes symbol or macro.
	RESET	Resets emulation processor.
	SAVE	Saves memory to file.
	SYMBOLS	Displays symbols.
	SUFFIX/BASE	Sets input and displays numeric base.
	TYPE	Sets/displays data type for symbol name.

Table 1. Summary of EMV-88 Commands (Continued)

Command Category	Command	Command Definition
Emulation Commands	BR	Displays breakpoint menu.
	BR0, BR1, BR2, BR3	Breakpoint register for execution address.
	BRB	Breaks on branch.
	BRR	Breakpoint register for execution range.
	BV	Breaks on value.
	BC	Clears all breaks.
	DTR	Displays trace menu.
	GO	Enters real-time emulation mode.
	MO	Break qualifier.
	PREVIOUS	Displays execution trace.
	STEP	Enters slow-down emulation mode.
	TD	Enable/disables display of code disassembly.
	TR	Enable/disables display of registers.
	TR0, TR1, TR2, TR3	Enable/disables display by execution address.
	TS	Enable/disables display of PSW.
	TV	Enable/disables display by register value.
Display/Modify Commands	ASM/DASM	Changes/displays code memory in assembly language mnemonics.
	ORG	Sets address for assembling instructions.
	DUMP	Displays memory as ASCII and hexadecimal.
	MEMORY	Displays menu for memory access.
	PORT	Changes/displays ports.
	REGISTER	Displays 8088 registers menu.
	BYTE WORD POINTER SINTEGER INTEGER	Change/display memory.
	REAL TREAL DREAL	8087 commands
Advanced Commands	DIR	Displays names of all available macros.
	FUNCTION	Invokes macro assigned to function key.
	MACRO	Displays macro text.
	MAP	Sets/displays memory map.
	PUT	Stores macro definitions.
	WRITE	Evaluates and displays expressions and strings.
	IF THEN COUNT REPEAT WHILE UNTIL	Control constructs.

intel®

APPLICATION NOTE

AP-262

October 1986

Using Procedures to Speed I^2ICE™ System Debugging

LAKSHMI JAYANTHI
DSO APPLICATIONS

Order Number: 280722-001

INTRODUCTION

Every engineering manager in charge of developing a microprocessor based product worries about the critical phase of the project when the hardware and software pieces are integrated. The integration portion of a development task can take as much as 40 percent of the design team's time, and some projects have been known to die at this point, succumbing to problems too expensive to solve.

Today's high-peformance 16-bit microprocessors make the integration phase more critical than ever. The move from 8-to 16-bit microprocessors has added considerable complexity to bus structures and memory management techniques. These processors address such large amounts of memory, the 80286 can access 16 megabytes of physical memory, that development of a product based on such a processor is software-intensive.

This additional complexity makes crucial the need for tools that can help hardware and software integrators gather, correlate, and evaluate data. Integrating the hardware and software of the microprocessor based system often ends in failure because of a classic communication gap between hardware and software designers. Each uses different methods to debug the system. The gap has been narrowed by such software development aids as an In-Circuit emulator (ICE™) system, PSCOPE, and TargetSCOPE-186 and such hardware tools as logic analyzers, but a full solution calls for a common interface that enables designers to go back and forth between hardware and software domains.

Such an interface is the I^{2}ICE™ (Integrated Instrumentation And In-Circuit Emulator) System. This new system integrates a logic analyzer, a high-level software debugger, and a sophisticated in-circuit emulator, permitting hardware and software developers to interact in the debugging process without learning each others language.

INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATOR (I^{2}ICE system) is a very powerful Hardware and Software debugging tool used to debug the hardware board or the software code. In order to achieve the debugging you need, numerous I^{2}ICE system features are available such as the following:

- Integrated Command Directory (ICD)
- Coprocessor Support
- Pseudo-variables
- Debug Procedures
- Logic Clips
- Logic Timing Analyzer

This application note focuses on the various ways to write debug procedures with an I^{2}ICE system for hardware and software debugging purposes. In general this note deals with procedures that can be used on an I^{2}ICE system. The numerous techniques that can be used in creating and developing these procedures are explained in detail with specific examples, flow charts, diagrams, displays, and actual procedure listings.

USEFUL DEFINITIONS

Before looking into the actual procedural language, it is extremely helpful to study some useful definitions. It is important to understand these definitions and their usage in the I^{2}ICE system in order to understand their role in the development of procedures. Each relevant command is defined and explained in some detail with examples. At the end of this section you will understand these definitions and their role in the procedural language.

Procedure

A procedure operates like a single command because it enables you to use several commands in a block structure and declare local variables. Also, the procedures can be several nested blocks. The size of procedures is limited only by the amount of memory space available. Defining procedures is like adding commands to the I^{2}ICE language and you can create commands of particular relevance to your debug situation.

INCLUDE

The INCLUDE command retrieves a command file from a mass storage device and loads it into memory.

LIST

A LIST file is an FICE system utility file. Typically, a list file is used as a debug session log.

LITERALLY

LITERALLY definitions are abbreviations for previously defined character strings.

WRITE

The WRITE command is most often used in procedures to add explanatory text to returned values in a more useful form, such as a table.

Attribute Codes

Attribute codes are used to add clarity or distinction to text written to the screen. These attribute codes are accessible using the CONCAT command. The CONCAT command builds strings by concatenating all or parts of old strings to form a new string.

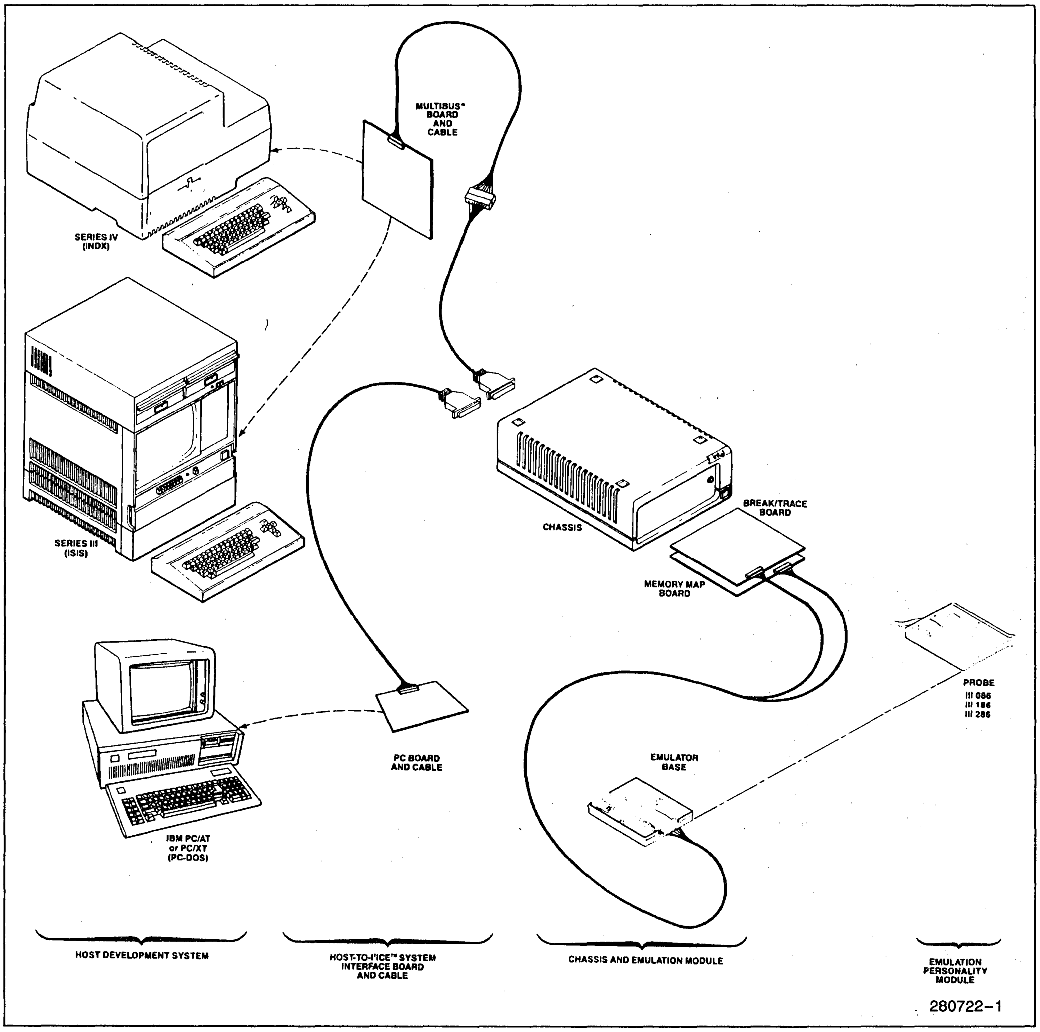

MULTIBUS* BOARD AND CABLE
SERIES IV (INDX)
SERIES III (ISIS)
IBM PC/AT or PC/XT (PC-DOS)
PC BOARD AND CABLE
CHASSIS
BREAK/TRACE BOARD
MEMORY MAP BOARD
EMULATOR BASE
PROBE III 086 III 186 III 286
HOST DEVELOPMENT SYSTEM
HOST-TO-I²ICE™ SYSTEM INTERFACE BOARD AND CABLE
CHASSIS AND EMULATION MODULE
EMULATION PERSONALITY MODULE
280722-1

Figure 1. I²ICE System

Boolean-Condition

A Boolean condition is either a value of type BOOLEAN (TRUE or FALSE) or an expression that uses one of the relational operators:

CI

The CI (console input) function enables a debug procedure to read one character from the system terminal.

IF

An IF command conditionally executes a command or group of commands.

PROBLEM DEFINITION

Now that we understand some of the relevant I²ICE definitions, we may move to the problem definition. All the preceding definitions will be used with examples of

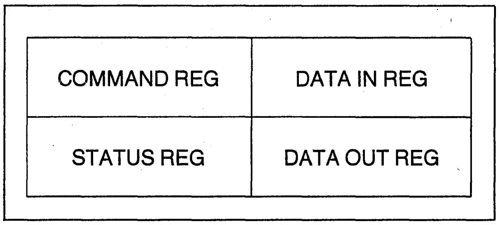


Figure 2. Registers in Asynchronous Mode

the procedural language. Debug procedures will be built-up in a variety of forms and their tremendous leverage and flexibility will be demonstrated. The countless number of ways commands like INCLUDE, LIST, WRITE are used and the numerous functions that they perform will be illustrated.

In this example we will debug a program which contains driver code interacting with peripheral components. The interaction with two of these peripherals (the simple 8251A Programmable Communications Interface and the more complex 82530 Serial Communications Controller) will be dealt with and also how these debug procedures can simplify and speed our work will be shown. We will develop techniques for working with these components and reduce frustration and constant thumbing through data catalogs.

The 8251A Programmable Communications Interface is a simple component as viewed by the software designer. It consists (in Asynchronous mode) of four registers as shown in Figure 2. When we read the status port, the multiple bits must be decoded to know what the 8251A is doing. Our first debug procedure will do this decoding for us and we will never confuse the bits again.

8251A Procedure

Prior to starting data transmission or reception, the 8251A must be loaded with a set of control words generated by the CPU. These control signals define the complete functional definition of the 8251A and must immediately follow a reset operation (internal or external). The control words are split into two formats: mode instruction and command instruction.

In data communications it is often necessary to examine the "status" of the active device to ascertain if errors have occurred or other conditions that require the processor's attention. With the 8251A facilities, the programmer can "read" the status of the device at any time during the functional operation.

The following procedure executes a status read of the 8251A and displays it on the I^{2}ICE system screen. The status read of the 8251A will be at some port location depending on the hardware configuration of the 8251A. This procedure, by way of example, requests the user to enter the port address of the 8251A from which the status information can be read. Operationally, you would define this once as a GLOBAL variable. The procedure then displays the byte value stored at that memory location in binary form on the screen. The corresponding messages for each of those eight bits with complete information about their status appears on the screen.

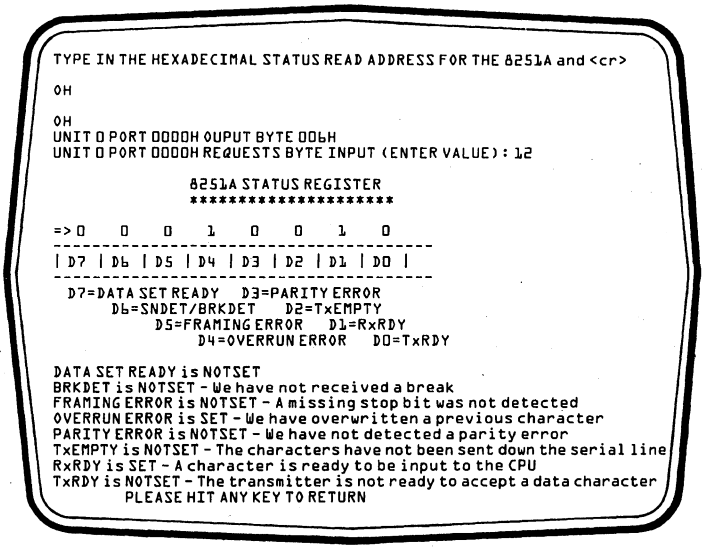
There are two procedures under one filename. They are GETHEX and I8251A, respectively. GETHEX is a procedure used to change the user input at the console to a hexadecimal value, to print this value on the screen, and to ring a bell if you make an improper selection. I8251A is the main procedure that reads in the hexadecimal value of the port address of the 8251A and (after you choose the port number) displays the eight bits of information in binary form with detailed explanation. The actual display on the I^{2}ICE System terminal screen is as shown in Figure 3.

```
*define PROC gethex = DO
*.define WORD num
*.define CHAR chr
*.define CHAR bell = 07h
*.num = 0
*.repeat
*..chr = ci
*..if (chr >= '0') and (chr <='9') then
*...num = num*10h + (chr-30h)
*...write using ('1,>') chr
*...else
*....if (chr >= 'A') and (chr <= 'F') then
*....num = num*10h + (chr-37h)
*....write using ('1,>') chr
*....else
*.....if (chr >= 'a') and (chr <= 'f') then
*.....num = num*10h + (chr-57h)
*.....write using ('1,>') chr
*.....else
*......if chr <> 0dh then write using('0,>') bell
*......end
*.....end
*....end
*...end
*...until chr == 0dh
*..end
*..return num
*.end
*define PROC I8251A = do
*.define WORD wreg
*.define WORD numget
*.define BYTE temp
*.curhome;cleareos
*.write concat(hibl,'ENTER THE PORT NUMBER OF THE 8251A and <cr>',norm)
*.wreg = gethex
*.temp = port(wreg)
*.curhome;cleareos
*.write concat(hi,'           8251A STATUS REGISTER',norm)
*.write concat(hi,'           ************************',norm)
*.write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
**(temp and 80h) /80h, &
**(temp and 40h) /40h, &
**(temp and 20h) /20h, &
**(temp and 10h) /10h, &
**(temp and 08h) /08h, &
**(temp and 04h) /04h, &
**(temp and 02h) /02h, &
**(temp and 01h) /01h
*.write concat(norm,' ________________________________________________')
*.write concat(norm,' | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | ')
*.write concat(norm,' ------------------------------------------------')
*.write '  D7 = DATA SET READY  D3 = PARITY ERROR'
*.write '     D6 = SYNDET/BRKDET  D2 = TxEMPTY'
*.write '        D5 = FRAMING ERROR  D1 = RxRDY'
*.write'            D4 = OVERRUN ERROR  D0 = TxRDY'
```

280722-2

```
*.if (temp and 80h)/80h = = 1 then
*..write'DATA SET READY is SET'
*..end
*.if (temp and 80h)/80h = = 0 then
*..write'DATA SET READY is NOTSET'
*..end
*.if (temp and 40h)/40h = =0 then
*..write'BRKDET is NOTSET — We have not received a break'
*..end
*.if (temp and 40h)/40h = = 1 then
*..write'BRKDET is SET — We have received a break'
*..end
*.if (temp and 20h)/20h = = 1 then
*..write'FRAMING ERROR is SET — A missing stop bit was detected'
*..end
*.if (temp and 20h)/20h = =0 then
*..write'FRAMING ERROR is NOTSET — A missing stop bit was not detected'
*..end
*..if (temp and 10h)/10h = = 1 then
*..write'OVERRUN ERROR is SET — We have overwritten a previous character'
*..end
*.if (temp and 10h)/10h = = 0 then
*..write'OVERRUN ERROR is NOTSET — We have not overwritten a previous character'
*..end
*.if (temp and 8h)/8h = =0 then
*..write'PARITY ERROR is NOTSET — We have not detected a parity error'
*..end
*.if (temp and 8h)/8h = = 1 then
*..write'PARITY ERROR is SET — We have detected a parity error'
*..end
*.if (temp and 4h)/4h = = 0 then
*..write'TxEMPTY is NOTSET — The characters have not been sent down the serial line'
*..end
*.if (temp and 4h)/4h = = 1 then
*..write'TxEMPTY is SET — All the characters have been sent down the serial line'
*..end
*.if (temp and 2h)/2h = =0 then
*..write'RxRDY is NOTSET — A character is not ready to be input to the CPU'
*..end
*.if (temp and 2h)/2h = = 1 then
*..write'RxRDY is SET — A character is ready to be input to the CPU'
*..end
*.if (temp and 1h)/1h = =0 then
*..write'TxRDY is NOTSET — The transmitter is not ready to accept a data character'
*..end
*.if (temp and 1h)/1h = = 1 then
*..write'TxRDY is SET — The transmitter is ready to accept a data character'
*..end
*.write concat(hibl,'                PLEASE HIT ANY KEY TO RETURN',norm)
*.temp=ci
*.end
```

280722-3



```
TYPE IN THE HEXADECIMAL STATUS READ ADDRESS FOR THE 8251A and <cr>
0H

0H
UNIT 0 PORT 0000H OUPUT BYTE 006H
UNIT 0 PORT 0000H REQUESTS BYTE INPUT (ENTER VALUE) : 12

                  8251A STATUS REGISTER
                  *********************

=> 0    0    0    1    0    0    1    0
-----------------------------------------
| D7  | D6  | D5  | D4  | D3  | D2  | D1  | D0  |
-----------------------------------------
  D7=DATA SET READY   D3=PARITY ERROR
      D6=SNDET/BRKDET    D2=TxEMPTY
           D5=FRAMING ERROR    D1=RxRDY
                D4=OVERRUN ERROR     D0=TxRDY

DATA SET READY is NOTSET
BRKDET is NOTSET - We have not received a break
FRAMING ERROR is NOTSET - A missing stop bit was not detected
OVERRUN ERROR is SET - We have overwritten a previous character
PARITY ERROR is NOTSET - We have not detected a parity error
TxEMPTY is NOTSET - The characters have not been sent down the serial line
RxRDY is SET - A character is ready to be input to the CPU
TxRDY is NOTSET - The transmitter is not ready to accept a data character
         PLEASE HIT ANY KEY TO RETURN
```

280722-4

Figure 3. 8251A Status Register Display

The 8251A is a very simple component. How could these techniques be used on a more complex chip such as the 82530 SCC? A procedures disk for the 82530 Serial Communications Controller (SCC) had been created (See Appendix B for availability). For those designs that do not utilize an 82530 SCC, it would still be beneficial to follow the text and deduce how the techniques developed could be used in your design. The purpose of creating this procedures disk is to minimize the usage of the 82530 Handbook. If you are using an 82530 chip in your design and also using an I^{2}ICE system to debug your design board, then all you need to do is invoke this set of procedures onto the I^{2}ICE system screen and all the required information will be given about the 82530. This drastically reduces the usage of the 82530 Handbook so that you can spend more time debugging and less hunting information.

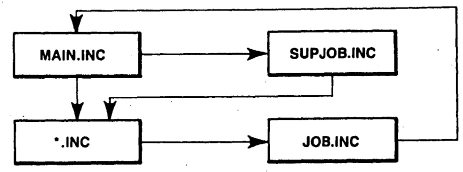
The different techniques that have been used in creating this disk are explained in detail in the following sections. Refer to Figure 4 which shows a flow diagram which describes the interaction of the four main blocks representing four different procedures. They are MAIN.INC, SUPJOB.INC, JOB.INC and *.INC where the '*' indicates any of the subset procedures. Every procedure built has two segments of that corresponding procedure. An overlay, as well as an include version, has been created.

The overlay (OVO) version contains the main contents of the material while the include (INC) file contains information about the procedure invocation, execution, and removal. The include file will be explained in a later section.

PROCEDURES DESCRIPTION

Procedures for the following pieces of information have been created from the 82530 Handbook.

Apart from the preceding set of procedures, some main procedures have also been created which are instrumental in managing the entire set of procedures. Those procedures are as follows:



280722-5

Figure 4. Flow Diagram

Handbook Description	Procedures Created	
• Pin Description	PINDES.OV0	PINDES.INC
• Register Function Description	REGDES.OV0	REGDES.INC
• Time Constant Values Description	TIMVAL.OV0	TIMVAL.INC
• Data Encoding Methods Description	DATENC.OV0	DATENC.INC
• Register Addressing Description	REGADD.OV0	REGADD.INC
• SCC Protocols Description	PRTCOL.OV0	PRTCOL.INC
• Register Bit Functions	PINFUN.OV0	PINFUN.INC

I2ICE.MAC—Initializes the drives and includes the main procedure, MAIN.OV0, and executes it.

MAIN.OV0—Handles the remainder of the procedural management. It contains the main core and acts as a supervisor.

MAIN.INC—Executes the procedure MAIN.OV0 and includes another procedure, JOB.INC.

JOB.INC—Used as a temporary storage file.

SUPJOB.INC—Used as a temporary storage file.

Each sub—procedure is loaded into memory when required and then removed following its execution. This technique saves memory, increases flexibility, and is faster in operation.

In the following sections we will look at each of these procedures in detail to understand the mechanism of operation and how they all fit into the conglomerate functionality picture.

82530 Procedure Explanation

COMMON FEATURES

In all the procedures that have been created uniformity has been maintained in certain definitions and functions. In all the procedures the following concepts and notations have been used:

- A message flashes on the screen at the beginning indicating that the procedures are "LOADING....".
- CURHOME and CLEAREOS are invoked after every major operation within the procedure.
- All the titles are in highlight code.
- All the console input requests are in blinking highlight code.
- A message in reverse video code appears on the screen in the end, displaying information on restarting the procedures.
- A BELL will ring if you make a selection that is not on that menu.
- One filename that contains more than one procedure, performs different functions interactively, but by itself performs a specific task.
- All procedures are removed when they are no longer required. This saves user memory space. (This is done by using the REMOVE command and saves user memory).
- If.... THEN....ELSE command statements test the different conditions within a procedure.
- Overlay files explain the main contents of the 82530 Handbook.
- Include files execute the corresponding procedures execution.
- All the local definitions are in the beginning of the procedure.

I2ICE.MAC

This is the marco file that has been created for a Series III/Series IV, with the I2ICE system software, a partial listing of this file is shown. This macro file resides in the same directory as the I2ICE system software. This macro file asks for the drive number that you have assigned for the procedures disk. The drive number is echoed onto the screen by the command "WRITE DEVICE". Depending on the drive selected, literals are defined for "INCLUDE" and "LIST" commands. At the end of the procedure the procedure MAIN.OV0 is invoked using the "INCLUDE" command and then executed. The temporary storage file, SUPJOB.INC, is also invoked using the "INCLUDE" command. The next menu tthat appears on the terminal is the main menu which is invoked by the procedure MAIN.OV0.

MAIN.OV0

This is the main operational procedure. All the other procedures are supervised by this procedure. Every time you are finished with a specific procedure, you are returned to the main menu. From the main menu you can go to the next menu screen or exit to the I2ICE system main interrogation menu if you so desire. Under the MAIN.OV0 filename, there are partitions into two smaller procedures, "W—COM" and "MAIN".

```
*define BOOLEAN flag = FALSE
*.define CHAR device
*.define GLOBAL BYTE drive
*.drive = 0
*.cury = 10t
*.write concat(hibl,'ON WHICH DRIVE IS THE PROCSDISK LOCATED?(0-9)',norm)
*.device = ci
*.write device
*.repeat while not flag
*.if device = = '0' then
*..drive = 0
*..define LITERALLY incc = 'include :f0'
*..define LITERALLY lst = 'list :f0'
*..end
*.flag = true
*.define LITERALLY i82530 = 'incc:main.inc nolist'
*.end
*incc:main.ov0 nolist
*main /*execute the procedure*/
*incc:supjob.inc nolist
```

280722-47

The procedure "W—COM" contains the LIST and NO-LIST files. If you want to exit from the procedures menu and enter the I²ICE system interrogation menu, then the message "SCC Procedures can be restarted by typing "I82530" appears on the screen in reverse video code.

However, if you make a choice from the main menu then the LIST command writes to SUPJOB.INC. The procedure corresponding to that particular choice is included. The overlay, as well the include portions of the procedure, are included. Then the NOLIST command closes the LIST file.

Note that in the W—COM procedure listing a parameter is being passed into the LIST file, SUBJOB.INC, using the WRITE USING option. The parameter that is being passed depends on the choice you make.

MAIN.INC

The procedure MAIN.INC contains the following two commands.

```
*main
*incc:supjob.inc nolist
```

280722-64

This procedure does nothing but execute its own procedure MAIN and include the temporary storage file, SUPJOB.INC. Note that SUPJOB.INC contains the include files of your choice. In the preceding example above we have chosen TIMVAL. Hence SUPJOB.INC contains the following two commands:

```
INCC:TIMVAL.OV0
INCC:TIMVAL.INC
```

This part of the procedure is the main part of the procedure MAIN.OV0 whose display is shown in Figure 5. In this part your selection appears on the terminal and your choice is echoed onto the screen. If you make a wrong choice, then a "Bell" rings. Depending on the choice made, the corresponding procedure is invoked. As mentioned previously that particular procedure is passed into SUPJOB.INC and executed.

Depending on the selection made by you that particular procedure is invoked.

Example:

If you choose to see the TIME CONSTANT VALUES description, then type in 3. The number 3 appears on the screen and the procedure name, TIMVAL, is passed into the procedure W—COM as a parameter. The procedure W—COM then includes the overlay as well as the include portions of the time constant value procedures using the LIST/NOLIST commands. If you choose the number 8 then you exit to the I²ICE system main menu.

```
*.chr = ci
*.if chr = = '3' then
*..write using ('0') chr
*..w_com('timval');return
*..if chr = = '8' then
*...write using ('0') chr
*...w_com('e');return
*...else
*...bell
*...end
*..end
```

280722-48

TIMVAL.OV0

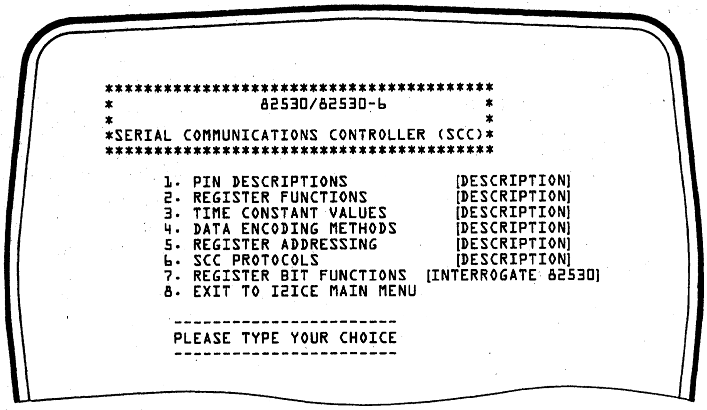
All the procedures that contain the 82530 descriptions are built containing the same format except for the pro-

```
*define PROC w_com = DO
*.curhome;cleareos
*.if %0 = = 'e' then
*..curhome;cleareos
*..write concat(revbl,'SCC Procedures can be restarted by typing"i82530")
*..norm
*..lst:supjob.inc
*..nolist
*..lst:job.inc
*..nolist
*..else
*..lst:supjob.inc
*..write using ("incc:",0,".ov0 nolist") %0
*..write using ("incc:",0,".inc nolist") %0
*..curhome;cleareos
*..nolist
*..end
*.end
```

280722-49

cedure describing REGISTER BIT FUNCTIONS, PINFUN.OVO. Hence let us consider the TIME CONSTANT VALUES DESCRIPTION procedure as an example to explain the techniques that have been used. The only procedure that is different, the REGISTER BIT FUNCTION procedure, will be explained later in this section.

TIMEVAL.OVO contains a message "LOADING. . ." that flashes on the terminal when this procedure is being included into the I²ICE system memory. The rest of the procedure contains the information contained in the 82530 Handbook. The WRITE command is used to type all the information on the screen. Towards the end of the procedure is a LIST command file. This LIST file lists the storage file, JOB.INC. The JOB.INC file contains the command that removes the procedure TIMVAL from the I²ICE system memory and invokes the procedure MAIN and also executes it so that you can choose a different option or return to the I²ICE system main menu. Note that I82530 is defined as a literal in the procedure I²ICE.MAC.



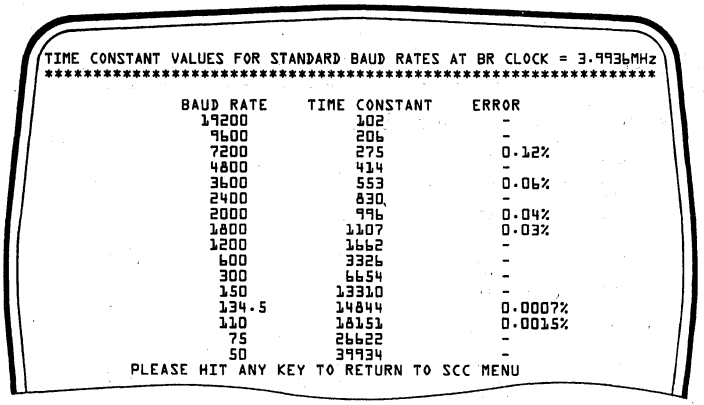
```
****************************************
*            82530/82530-6             *
*                                      *
*SERIAL COMMUNICATIONS CONTROLLER (SCC)*
****************************************

      1. PIN DESCRIPTIONS           [DESCRIPTION]
      2. REGISTER FUNCTIONS         [DESCRIPTION]
      3. TIME CONSTANT VALUES       [DESCRIPTION]
      4. DATA ENCODING METHODS      [DESCRIPTION]
      5. REGISTER ADDRESSING        [DESCRIPTION]
      6. SCC PROTOCOLS              [DESCRIPTION]
      7. REGISTER BIT FUNCTIONS  [INTERROGATE 82530]
      8. EXIT TO I2ICE MAIN MENU

       -----------------------
       PLEASE TYPE YOUR CHOICE
       -----------------------
```

280722-6

Figure 5. Main Menu Register

```
*define PROC timval = DO
*.write concat(hibl,'LOADING....')
*.norm
*.curhome;cleareos
*.write concat(hi,'TIME CONSTANT VALUES FOR STANDARD BAUD RATES AT BR CLOCK = 3.9936MHz')
*.write concat(hi,'********************************************************************************')
*.norm
*.write '      BAUD RATE      TIME CONSTANT      ERROR'
*.write '        19200             102              -'
*.write '         9600             206              -'
*.write '         7200             275            0.12%'
*.write '         4800             414              -'
*.write '         3600             553            0.06%'
*.write '         2400             830              -'
*.write '         2000             996            0.04%'
*.write '         1800            1107            0.03%'
*.write '         1200            1662              -'
*.write '          600            3326              -'
*.write '          300            6654              -'
*.write '          150           13310              -'
*.write '          134.5         14844            0.0007%'
*.write '          110           18151            0.0015%'
*.write '           75           26622              -'
*.write '           50           39934              -'
*.write concat(hibl,'            PLEASE HIT ANY KEY TO RETURN TO SCC MENU')
*.norm
*.define CHAR yyy = ci
*.lst:job.inc
*.write 'remove timval'
*.write 'i82530'
*.nolist
*.curhome;cleareos;end
```

280722-7



```
TIME CONSTANT VALUES FOR STANDARD BAUD RATES AT BR CLOCK = 3.9936MHz
*******************************************************************

           BAUD RATE     TIME CONSTANT     ERROR
             19200           102             -
              9600           206             -
              7200           275           0.12%
              4800           414             -
              3600           553           0.06%
              2400           830             -
              2000           996           0.04%
              1800          1107           0.03%
              1200          1662             -
               600          3326             -
               300          6654             -
               150         13310             -
             134.5         14844           0.0007%
               110         18151           0.0015%
                75         26622             -
                50         39934             -
        PLEASE HIT ANY KEY TO RETURN TO SCC MENU
```

280722-8

Figure 6. Time Constant Values Description

TIMVAL.INC

The procedure TIMVAL.INC contains the following two commands:

```
*timval
*incc:job.inc nolist
```

280722-65

This procedure, as well as all the other .INC procedures contains the preceding two commands. The first command executes that specific procedure and the next command includes the storage procedure JOB.INC. As mentioned previously the procedure JOB.INC removes that particular procedure from the I²ICE system memory and also invokes and executes the main procedure.

PINFUN.OVO

This is the procedure that explains the register bit functions of the 82530 Serial Communications Controller. This procedure is very involved and performs all the functions that are required by the 82530 handbook. This procedure, PINFUN.OVO, is divided into a subset of procedures contained within the same file #. The following sections explain the different interactive procedures and their functions, respectively.

GETNUM—This is a procedure that changes the number entered at the console into decimal base. If an inappropriate number is input then the bell rings. The decimal base number is returned to the procedure that calls it. The console inut is echoed onto the screen.

```
*define PROC getnum = DO
*.define BYTE num
*.define CHAR chr
*.define CHAR bell = 07h
*.num = 0
*.repeat
*..chr = ci
*..if (chr > = '0') and (chr < = '9') then
*...num = num*10t + (chr-30h)
*...write using ('1,> ') chr
*...else
*...if chr < > 0dh then write using('0,> ') bell
*....end
*...end
*..until chr = = 0dh
*..end
*.return num
*.end
```

280722-50

GETP1—This procedure sets the I/O ports for the I²ICE system as well as the starting addresses for the 82530. If you have not mapped the I/O ports, then the method to port is explained on the screen along with menu prompts so that type only the starting address and number of bytes either mapped to ICE memory or USER memory. This procedure also requests your input for the hexadecimal starting address for the 82530.

```
*define PROC getpl = DO
*.curhome;cleareos
*.define CHAR zzz
*.write concat(hi,'MAPIO COMMAND DISPLAYS OR SETS PHYSICAL LOCATION FOR I/O PORTS')
*.write concat(norm,'',hibl)
*.write using('''          HAVE YOU MAPPED THE I/O PORTS?(Y/N)'',>') chrin
*.norm
*.repeat
*..chrin = ci
*..if not((chrin = = 'N') or (chrin = = 'n') or (chrin = = 'Y') or (chrin = = 'y')) then
*...bell
*...endif
*..until (chrin = = 'N') or (chrin = = 'n') or (chrin = = 'Y') or (chrin = = 'y')
*..end
*.if (chrin = = 'N') or (chrin = = 'n') then do
*..write using('0') chrin
*..curhome;cleareos
*..write '         THE I/O PORTS CAN BE MAPPED THE FOLLOWING WAY'
*..write ' '
*..write '              MAPIO [(partition) <USER or ICE>]'
*..write ''
*..write '    MAPIO — displays the current map of I/O port address blocks'
*..write ''
*..write ' partition — is an entry specifying a range of addresses such as:'
*..write '              Starting port-address LENGTH number of bytes'
*..write ''
*..write '    USER — Maps I/O to the user system'
*..write ''
*..write '    ICE — Maps I/O to the I²ICE probe'
*..write ''
*..write concat(hibl,'PLEASE TYPE MAPIO STARTING HEXADECIMAL PORT-ADDRESS and <cr>')
*..norm
*..paddr = gethex
*..write 'H'
*..write concat(hibl,'    PLEASE TYPE NUMBER OF BYTES and <cr>')
*..norm
*..paddr1 = gethex
*..write 'H'
*..write concat(hibl,' PLEASE TYPE "U" FOR USER or "I" FOR ICE')
*..norm
*..repeat
*..zzz = ci
*..if not((zzz = = 'U') or (zzz = = 'u') or (zzz = = 'I') or (zzz = = 'i')) then
*...bell
*...endif
*...until (zzz = = 'U') or (zzz = = 'u') or (zzz = = 'I') or (zzz = = 'i')
*..end
*..if (zzz= ='I') or (zzz= ='i') then
*..MAPIO PADDR LENGTH PADDR1 ICE
*..write using("MAPIO ",0,H," LENGTH ",0,H," ICE "')paddr,paddr1
*..write ''
*..write concat(hibl,'              PLEASE HIT ANY KEY TO CONTINUE')
*..norm
*..zzz = ci;else
*..if (zzz= ='U') or (zzz= ='u') then
*...MAPIO PADDR LENGTH PADDR1 USER
*...write using("MAPIO ",0,H," LENGTH ",0,H," USER "')paddr,paddr1
*...write ''
*...write concat(hibl,' PLEASE HIT ANY KEY TO CONTINUE')
*...norm
```

280722-9

```
*...chrin = ci;endif;end
*...end
*..end
*.curhome;cleareos
*.write concat(hibl,'TYPE IN THE HEXADECIMAL ADDRESS FOR THE 82530 and <cr>')
*.norm
*.wreg = gethex
*.write 'H'
*.write ''
*.end
```

GETP2 — This procedure asks for console input on your choice of whether you want to read a register, or write to one, or change the 82530 port address, or return to the main menu.

```
*define PROC getp2 = DO
*.curhome;cleareos
*.write concat(hi,'          ************************')
*.write concat(hi,'          *PIN FUNCTION MENU*')
*.write concat(hi,'          ************************')
*.norm
*.write ' '
*.write ' '
*.write '          "R" — READ A REGISTER'
*.write '          "W" — WRITE TO A REGISTER'
*.write '          "N" — CHANGE 82530 PORT ADDRESS'
*.write '          "E" — EXIT TO MAIN MENU'
*.write ' '
*.write ' '
*.write concat(hibl,'          PLEASE TYPE YOUR CHOICE')
*.norm
*.end
```

GETP3 — This procedure gives the choices of the READ and WRITE REGISTERS that are available for selection and asks for console input.

```
*define PROC getp3 = DO
*.curhome;cleareos
*.write ' '
*.write concat(hi,'  ********************************************************')
*.write concat(hi,'  THE CHOICES OF REGISTERS FOR YOUR SELECTION ARE :')
*.write concat(hi,'  ********************************************************')
*.norm
*.if (chrin = = 'R') or (chrin = = 'r') then
*..write '     0. RR0          1. RR1          2. RR2'
*..write '     3. RR3          8. RR8         10. RR10'
*..write '    12. RR12        13. RR13        15. RR15'
*..else
*..write '     0. WR0          1. WR1          2. WR2'
*..write '     3. WR3          4. WR4          5. WR5'
*..write '     6. WR6          7. WR7          8. WR8'
*..write '     9. WR9         10. WR10        11. WR11'
*..write '    12. WR12        13. WR13        14. WR14'
*..write '                    15. WR15  '
*..end
*.write ' '
*.write ' '
*.write concat(hibl,'          PLEASE TYPE YOUR CHOICE and <cr>')
*.norm
```

280722-10

```
*.regnum = getnum
*.write 'T'
*.write ''
*.end
```

280722-66

GETMEN1—This procedure accepts your choice for the pin function menu and, depending on the choice you have made, calls the appropriate procedure. If you want to read or write to a register then the procedure GETP3, which displays the read and write register selection is invoked. On the other hand, if you want to change the 82530 port address then the procedure GETP1, which maps the I/O ports of the I^{2}ICE system, is invoked.

```
*define PROC getmen1 = DO
*.if check then
*..getp1
*..check = false
*..end
*.repeat
*.getp2
*.chrin = ci
*.if (chrin = = 'E') or (chrin = = 'e') then write using ('0') chrin;return end
*..if (chrin = = 'R') or (chrin = = 'r') or (chrin = = 'W') or (chrin = = 'w') then
*..write using ('0') chrin
*..curhome;cleareos
*..getp3
*..return
*..else
*..if (chrin = = 'N') or (chrin = = 'n') then
*...write using ('0') chrin
*...getp1
*...else
*...if (chrin = = 1bh) then return end
*....bell
*....end
*...end
*..end
*.end
```

280722-51

W_COM1_This procedure uses LIST/NOLIST as well as INCLUDE and REMOVE to perform numerous functions. Once the usage of these procedures is complete, all the corresponding procedures are removed using the REMOVE command listed in the procedure JOB.INC and the main menu is invoked. If you choose to read or write to a particular register then that particular procedure containing the register's contents is included. After the register is read or written into, then this procedure removes this particular procedure from the I^{2}ICE system memory and re-invokes the register bit function menu for you to make the next selection.

```
*define PROC w_com1 = DO
*.curhome;cleareos
*..if %0 = = 'e' then
*..lst:job.inc
*..write 'remove done,gethex,getnum,getmen1,w_com1,getp1,getp2,getp3'
*..write 'remove chrin,check,wreg,regnum,paddr,paddr1'
*..write 'i82530'
*..nolist
*..else
*..lst:job.inc
*..write using ("incc:",0," nolist") %0
*..write using ('0') %0
*..write using ("remove ",0') %0
*..write 'incc:pinfun.inc nolist'
*..nolist
*..end
*.end
```

280722-52

DONE—This procedure is the main body of the file PINFUN.OV0. This is the nucleus which controls the operations of all the other procedures within this filename and inter-acts with them. Depending on your selection, the procedure W__COM1 is executed with the corresponding register value passed as a parameter. If a wrong selection is made, then the message "UNABLE TO READ REGISTER" appears on the screen. Your choice is echoed onto the screen. An inappropriate selection will ring the bell. For example, if you want to read a register "RR0" invoke the procedure W__COM1 and pass the read register procedure RR0 as a parameter as shown in the following listing. Similarly if you want to write to register "WR0", invoke the procedure.

```
*define PROC done = DO
*.curhome ; cleareos
*.getmen1
*..if (chrin = = 'E') or (chrin = = 'e') then
*..w__com1('e')
*..return
*..end
*..repeat
*...if (chrin = = 'R') or (chrin = = 'r') then
*...do
*....if regnum = = 0 then
*....w__com1('rr0') ; return
*....end
*...getp3
*...end
*...if (chrin = = 'W') or (chrin = = 'w') then
*...do
*...if regnum = = 0 then
*...w__com1('wr0'); return
*...end
*...getp3
*...end
*..end
*.end
```

280722-53

PINFUN.INC

This procedure contains the following two commands:

```
*done
*incc:job.inc nolist
```

280722-67

This procedure executes the main procedure under the filename PINFUN.OV0, DONE, and then includes the temporary storage file, JOB.INC.

RR0

This procedure reads a value from the console which is written into WREG. This procedure also displays this read value in binary form and explains what each of these bits correspond to in an actual 82530.

```
*curhome;cleareos
*write concat(hibl,'LOADING....')
*norm
*define PROC rr0 = DO
*.define CHAR yyy
*.define BYTE temp
*.temp = port(wreg)
*.curhome ;cleareos
*.write concat(hi,' READ REGISTER 0')
*.write concat(hi,' *****************')
```

280722-54

```
*.norm
*.write using ('2c,"= >",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
*.(temp and 80h) /80h, &
*.(temp and 40h) /40h, &
*.(temp and 20h) /20h, &
*.(temp and 10h) /10h, &
*.(temp and 08h) /08h, &
*.(temp and 04h) /04h, &
*.(temp and 02h) /02h, &
*.(temp and 01h) /01h
*.write concat(norm,'  ________________________________________')
*.write concat(norm,'  | D7| D6| D5| D4| D3| D2| D1| D0| ')
*.write concat(norm,'  ----------------------------------------')
*.write '    D7 = BREAK/ABORT'
*.write '       D6 = Tx UNDERRUN/EOM'
*.write '          D5 = CTS'
*.write '             D4 = SYNC/HUNT'
*.write '                D3 = CD'
*.write '                   D2 = Tx BUFFER EMPTY'
*.write '                      D1 = ZERO COUNT'
*.write '                         D0 = Rx CHAR. AVAIL.'
*.write ' '
*.write concat(hibl,' PLEASE HIT ANY KEY TO RETURN')
*.norm
*.yyy = ci
*.end
```

280722-11

WR0

This procedure writes to a register from the console. The eight different bits in the 82530 write register are explained and the console requests for an input from you.

```
*curhome;cleareos
*write concat(hibl,'LOADING....',norm)
*define PROC wr0 = DO
*.define CHAR yyy
*.define BYTE temp
*.curhome;cleareos
*.write concat(hi,' WRITE REGISTER 0'
*.write concat(hi,' ******************'
*.write concat(norm,'  ________________________________________')
*.write concat(norm,'  | D7| D6| D5| D4| D3| D2| D1| D0| ')
*.write concat(norm,'  ----------------------------------------')
*.write '           D7  D6'
*.write '              0 0 = NULL CODE '
*.write '              0 1 = RESET Rx CRC CHECKER'
*.write '              1 0 = RESET Tx CRC GENERATOR'
*.write '              1 1 = RESET Tx UNDERRUN/EOM LATCH'
*.write ' D5 D4 D3                                   D2 D1 D0'
*.write ' 0  0  0  = NULL CODE                       0  0  0  = 0 or 8'
*.write ' 0  0  1  = POINT HIGH REGISTER GROUP       0  0  1  = 1 or 9'
*.write ' 0  1  0  = PRESET EXT/STATUS INTERRUPTS    0  1  0  = 2 or 10'
*.write ' 0  1  1  = SEND ABORT                      0  1  1  = 3 or 11'
*.write ' 1  0  0  = ENABLE INT ON NEXT Rx CHAR.     1  0  0  = 4 or 12'
*.write ' 1  0  1  = RESET TxINT PENDING             1  0  1  = 5 or 13'
*.write ' 1  1  0  = ERROR RESET                     1  1  0  = 6 or 14'
*.write ' 1  1  1  = RESET HIGHEST IUS               1  1  1  = 7 or 15'
*.write concat(hibl,'PLEASE TYPE THE VALUE TO BE WRITTEN and <cr> ',norm)
*.temp = gethex
```

280722-12

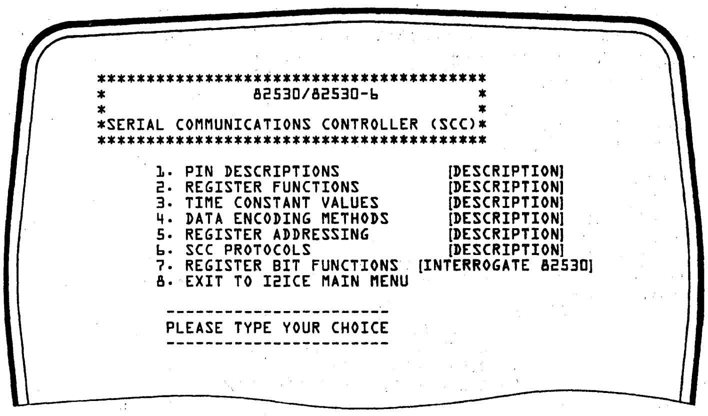
```
1457: *.port(wreg) = 0
1458: *.port(wreg) = temp
1460: *.write concat(hibl,'
1462: *.yyy = ci
1463: *.end
```

280722-68

Example

Let us go through the entire menu with a specific example.

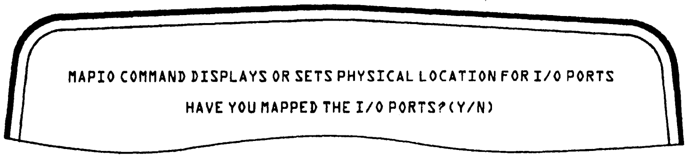
- Select "REGISTER BIT FUNCTION". (Figure 7)
- Select N for "HAVE YOU MAPPED THE I/O PORTS?" (Figure 8)
- Select 0 for "PLEASE TYPE MAPIO HEXADECIMAL PORT ADDRESS and <cr>". (Figure 8)
- Select 40 for "PLEASE TYPE NUMBER OF BYTES and <cr>" (Figure 8)
- Select I for "PLEASE TYPE 'U' FOR USER OR 'I' FOR ICE" (Figure 9)
- Select 0 for "TYPE IN THE HEXADECIMAL ADDRESS FOR THE 82530 and <cr>". (Figure 10)
- Select R for READ A REGISTER (Figure 11)
- Select 0 for "THE CHOICES OF REGISTER FOR YOUR SELECTION ARE:" (Figure 12)
- Select E to "EXIT TO THE MAIN MENU". (Figure 13)
- Select E to "EXIT TO THE I²ICE MAIN MENU". (Figure 14)



```
*****************************************
*             82530/82530-6             *
*                                       *
*SERIAL COMMUNICATIONS CONTROLLER (SCC)*
*****************************************

        1. PIN DESCRIPTIONS          [DESCRIPTION]
        2. REGISTER FUNCTIONS        [DESCRIPTION]
        3. TIME CONSTANT VALUES      [DESCRIPTION]
        4. DATA ENCODING METHODS     [DESCRIPTION]
        5. REGISTER ADDRESSING       [DESCRIPTION]
        6. SCC PROTOCOLS             [DESCRIPTION]
        7. REGISTER BIT FUNCTIONS  [INTERROGATE 82530]
        8. EXIT TO I2ICE MAIN MENU

         -----------------------
         PLEASE TYPE YOUR CHOICE
         -----------------------
```

280722-13

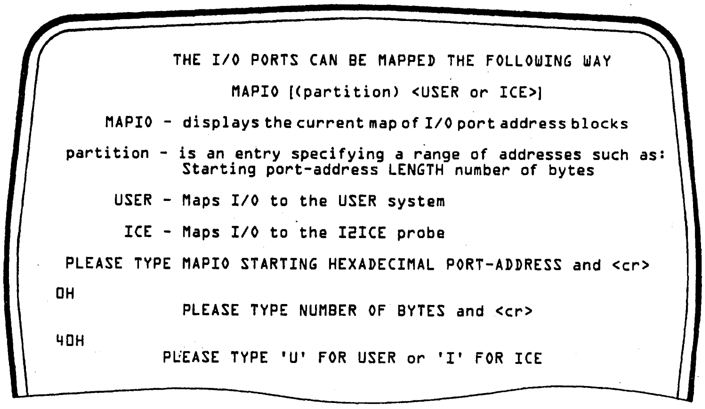
Figure 7. Main Menu Display



```
MAPIO COMMAND DISPLAYS OR SETS PHYSICAL LOCATION FOR I/O PORTS
          HAVE YOU MAPPED THE I/O PORTS?(Y/N)
```

280722-14

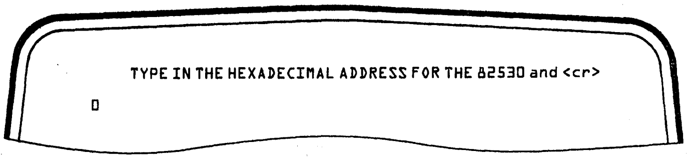
Figure 8. I/O Port Display



```
          THE I/O PORTS CAN BE MAPPED THE FOLLOWING WAY
               MAPIO [(partition) <USER or ICE>]
    MAPIO - displays the current map of I/O port address blocks
partition - is an entry specifying a range of addresses such as:
            Starting port-address LENGTH number of bytes
     USER - Maps I/O to the USER system
      ICE - Maps I/O to the I2ICE probe
PLEASE TYPE MAPIO STARTING HEXADECIMAL PORT-ADDRESS and <cr>
0H
           PLEASE TYPE NUMBER OF BYTES and <cr>
40H
         PLEASE TYPE 'U' FOR USER or 'I' FOR ICE
```

280722-15

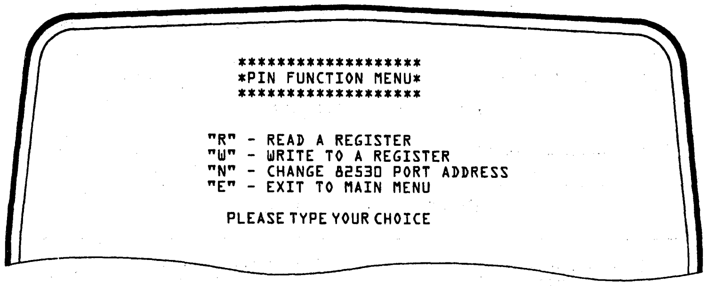
Figure 9. MAPIO Conditions



```
    TYPE IN THE HEXADECIMAL ADDRESS FOR THE 82530 and <cr>
0
```

280722-16

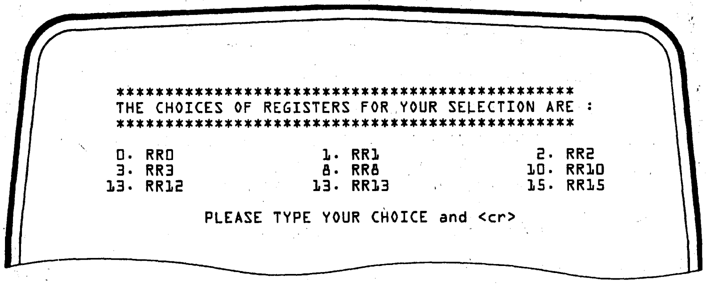
Figure 10. Hexadecimal Port Address



```
*******************
*PIN FUNCTION MENU*
*******************

"R" - READ A REGISTER
"W" - WRITE TO A REGISTER
"N" - CHANGE 82530 PORT ADDRESS
"E" - EXIT TO MAIN MENU

 PLEASE TYPE YOUR CHOICE
```

280722-17

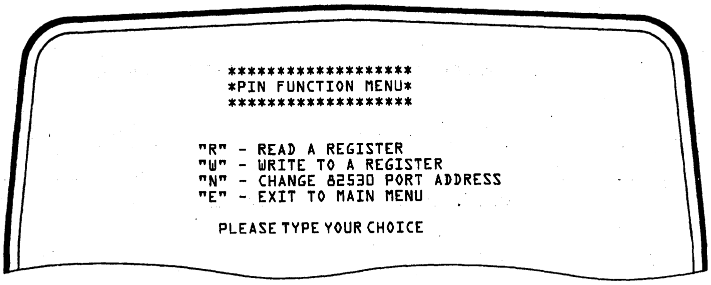
Figure 11. Pin Function Menu



```
************************************************
THE CHOICES OF REGISTERS FOR YOUR SELECTION ARE :
************************************************

 0. RR0                1. RR1                2. RR2
 3. RR3                8. RR8               10. RR10
13. RR12              13. RR13              15. RR15

        PLEASE TYPE YOUR CHOICE and <cr>
```

280722-18

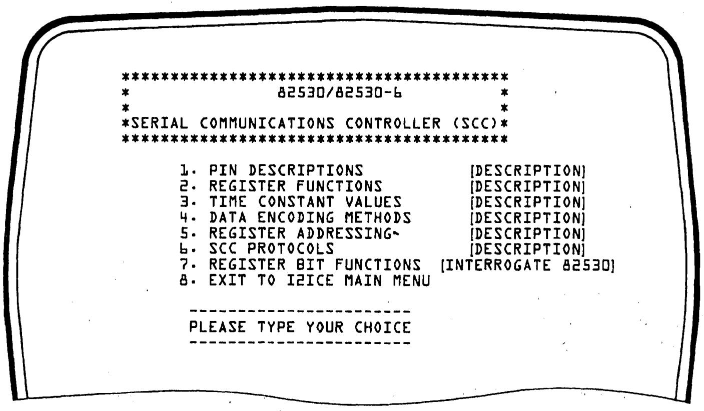
Figure 12. Register Selections



```
*******************
*PIN FUNCTION MENU*
*******************

"R" - READ A REGISTER
"W" - WRITE TO A REGISTER
"N" - CHANGE 82530 PORT ADDRESS
"E" - EXIT TO MAIN MENU

 PLEASE TYPE YOUR CHOICE
```

280722-19

Figure 13. Pin Function Menu



```
*****************************************
*              82530/82530-6            *
*                                       *
*SERIAL COMMUNICATIONS CONTROLLER (SCC)*
*****************************************

        1. PIN DESCRIPTIONS         [DESCRIPTION]
        2. REGISTER FUNCTIONS       [DESCRIPTION]
        3. TIME CONSTANT VALUES     [DESCRIPTION]
        4. DATA ENCODING METHODS    [DESCRIPTION]
        5. REGISTER ADDRESSING      [DESCRIPTION]
        6. SCC PROTOCOLS            [DESCRIPTION]
        7. REGISTER BIT FUNCTIONS  [INTERROGATE 82530]
        8. EXIT TO I2ICE MAIN MENU

         -----------------------
         PLEASE TYPE YOUR CHOICE
         -----------------------
```

280722–20

Figure 14. Main Menu

CONCLUSION

The trend in microprocessor instrumentation is toward complete integration of hardware and software design and test components. Debugging systems, system design, and program control utilities will be integrated into one development tool designed to bring users closer to a completely virtual engineering environment.

The I^{2}ICE system combines in-circuit emulation, high-level language software debugging and logic analysis in a system designed to improve productivity in the development and integration of complex microprocessor systems. One of the major features of an I^{2}ICE system is the usage of procedures. Given an I^{2}ICE system, procedures can be written to debug any peripheral chip on the target system by building procedures to simulate the chip and to communicate with it through the I^{2}ICE system. This enhances the debugging capabilities of I^{2}ICE system.

This application note has illustrated the numerous features of procedural implementation using a very simple peripheral component, 8251A, and a complicated component such as the 82530. We have shown that the I^{2}ICE system is able to debug procedures as well as standard microprocessor code. Now you can write your procedures for any peripheral component and hence aid in debugging your design using the I^{2}ICE system. You will spend less time in referring to the data books and more time in your design. This will help to deliver your product on schedule.

APPENDIX A GLOSSARY

PROCEDURE

A procedure operates like a single command because it enables you to use several commands in a block structure and declare local variables. Also, the procedures can be several nested blocks. The size of procedures is limited only by the amount of memory space available. Defining procedures is like adding commands to the I²ICE language and you can create commands of particular relevance to your debug situation.

Although a debug procedure is not executed until its name is invoked, the I²ICE system checks the syntax when the procedure is defined and determines the type of all referenced objects. Changing the type and/or definition of an object in the procedure before it is executed can cause errors when the procedure is executed.

Procedures can be defined within other procedures. The inner procedure is not visible to the I²ICE system until the outer procedure is executed. Once procedures become visible to the system, they are always global, even when nested inside other procedures. A debug procedure cannot forward reference a debug object. Values can be returned from the procedures, and parameters can be referenced in procedures. The RETURN command is used to return the procedure values to the terminal.

The percent sign (%) signals the I²ICE system to expect an expression.

%NP is a predefined system parameter equal to the number of parameters passed in the debug procedure.

%Number defines a parameter number which selects that parameter from the list following the debug procedure invocation. Numbers range consecutively from 0 to 99.

%(Expression) is used instead of a number but requires parentheses. The expression must evaluate to a number between 0 and 99.

The slash asterisk combination (/*..........*/) defines a comment.

Example:

In the following example we will define an averaging procedure. Data is supplied by the parameter list. This procedure returns the average value of all the parameters.

```
*DEFINE PROC average = DO    /*Define the debug procedure*/
.*DEFINE INTEGER sum = 0
.*DEFINE BYTE I = 0     /*Initialize variables*
.*Count %NP             /*Count is equal to the # of parameters*/
..*sum = sum + %(I)     /*Add I to the sum*/
...*I = I + 1           /*Increment I*/
..*ENDCOUNT             /*Defines the extant of the COUNT loop*/
.*RETURN sum/%NP        /*Return the parameter average*/
.*END
```

Enter the following command to display the debug procedure on the terminal screen.

```
*PROC average                /*Display the debug procedure definition*/

define proc AVERAGE = do
define integer SUM = 0
define byte I = 0
count %NP
SUM = SUM + %(I)
I = I + 1
endcount
return
SUM/%NP
end
```

The following command executes this debug procedure to find the average of the three numbers.

```
* AVERAGE (4, 5, 21T)                  /*Execute the debug procedure*/
+10
```

But this is only a simple example. I²ICE contains many features which enable us to produce comprehensive procedures. Features such as INCLUDE, LIST, LITERALLY, WRITE, BOOLEAN CONDITIONS and much more.

INCLUDE

The INCLUDE command retrieves a command file from a mass storage device and loads it into memory.

Command files may be created in two ways: by creating a file with the integrated I²ICE screen editor or by saving definitions created during a debug session to a file with the PUT or APPEND command.

Note that INCLUDE has the following restrictions:

- You can nest INCLUDE commands (limited by available memory), but they must be the last item on a line.
- An INCLUDE command cannot appear in block structures (i.e., REPEAT, COUNT, IF, DO/END, or a debug procedure).
- Input cannot originate at the terminal (i.e., INCLUDE :CI:)
- The INCLUDE command must be the last command on the line.

LIST

A LIST file is an I²ICE system utility file. Typically, a list file is used to create a permanent log of a debug session. All interactions between the I²ICE system and the terminal (except edits) are recorded in an open LIST file. Only one list file can be opened at a time. The list files can be closed by issuing the NOLIST command or by terminating a debugging session.

Example:

The following commands open and close a LIST file to record the debug session.

```
*LIST UL16.85
*<Record the debug session>
*NOLIST
```

LITERALLY

LITERALLY definitions are abbreviations for previously defined character strings. LITERALLY definitions save keystrokes and improve clarity. For example, the following definition saves three keystrokes. Once this LITERALLY is defined you can type DEF for DEFINE.

```
*DEFINE LITERALLY DEF = 'DEFINE'
```

These definitions may be saved to disk and auto-reloaded. In addition, an automatic LITERALLY expansion feature can be turned on and off. The examples that follow use this LITERALLY feature extensively.

WRITE

The WRITE command is often used in procedures to add explanatory text to returned values in a more useful form, such as a table.

The WRITE command displays a maximum of 200 bytes of data. Unless specified in the format string by the continuation symbol (&), the information in the write buffer is deleted at the end of every write. If the write-list contains more items than are specified by the format string, the format string is reused from the beginning, until all write-items are displayed according to the format.

Example:

The following example shows the WRITE USING option. The procedure SQUAREIT squares a number specified when the procedure is called (%0).

```
*DEFINE    PROC    squareit = DO
.*WRITE      USING( "The square of ",X,T,0,X,"is",X,T,0,')%0,%0*%0
.*END
```

Call the procedure and specify the number to be squared.

```
*squareit(7)
The square of 7 is 49
```

ATTRIBUTE CODES

Attribute codes are used to add clarity or distinction to text written to the screen. These attribute codes are used with the CONCAT command. The CONCAT command builds strings by concatenating all or part of old strings to form a new string. Placing a CONCAT function inside of a debug procedure saves the construction and prints it when the procedure is executed. The various codes that are available on a host system are:

Attribute Codes	Series III/Series IV	IBM-PC
Blinking Code	82II	5
Reverse Video	90H	7
Highlight Code	81H	1
Blinking/Reverse Video	92H	7;5
Blinking/Highlight	83H	1;5
Normal	80GH	0

Example:

In this example "Hello!" is written in blinking code and the screen returns to the normal attribute code immediately afterwards. The following steps perform these functions. Note that the rest of the screen will stay in the specified attribute mode until the mode is changed or turned off.

SeriesIII/SeriesIV

```
*define global CHAR esc = 1bh
*define global CHAR blink = concat (esc, 'L',82H)
*define global CHAR norm = concat(esc, 'L',80H)
*write concat(blink, 'Hello', norm) /*BLINKING CODE*/
 Hello!
```

IBM PC XT/AT

```
*define global CHAR esc = 1bh
*define global CHAR blink = concat (esc,4eh,esc,"[0m',esc,"[5m')
*define global CHAR norm = concat(esc,"[0m",esc,4eh)
*write concat(blink, "Hello! "',norm) /*BLINKING CODE*/
Hello!
```

BOOLEAN-CONDITION

A Boolean condition is either a value of type BOOLEAN (TRUE or FALSE) or an expression that uses one of the following relational operators:

= = equal to
> greater than
< less than
> = greater than or equal to
< = less than or equal to
< > not equal to

CI-

The CI (console input) function enables a debug procedure to read one character from the system terminal. The procedure pauses until the character is entered. No prompt is displayed while the system is waiting for the CI character, and the entered character is not echoed to the screen. No carriage return is required after the character has been keyed in.

IF

An IF command conditionally executes a command or group of commands. Debug objects are local only in memory type definitions and DO/END blocks. Literals and debug procedures are always global.

Syntax:

If boolean-condition THEN
[I^{2}ICE commands]
[ELSE[I^{2}ICE commands]]
END[IF]

APPENDIX B PROCEDURES LISTINGS

```
                                    *I2ICE.MAC*
                                    ***********

define BOOLEAN flag = FALSE
define CHAR device
define GLOBAL BYTE drive
drive = 0
cury = 10t
write concat(1bh,'L',83h), '          ON WHICH DRIVE IS THE PROC:DISK LOCATED?(0 - 9) '
write concat(1bh,'L',80h),''
device = ci
write device
repeat while not flag
if device=='0' then
drive=0
define LITERALLY incc='include :f0'
define LITERALLY lst = 'list :f0'
else
if device=='1' then
drive=1
define LITERALLY incc='include :f1'
define LITERALLY lst = 'list :f1'
else
if device=='2' then
drive=2
define LITERALLY incc='include :f2'
define LITERALLY lst = 'list :f2'
else
if device=='3' then
drive=3
define LITERALLY incc='include :f3'
define LITERALLY lst = 'list :f3'
else
if device=='4' then
drive=4
define LITERALLY incc='include :f4'
define LITERALLY lst = 'list :f4'
else
if device=='5' then
drive=5
define LITERALLY incc='include :f5'
define LITERALLY lst = 'list :f5'
else
if device=='6' then
drive=6
define LITERALLY incc = ' include :f6'
define LITERALLY lst = 'list :f6'
else
if device=='7' then
drive=7
define LITERALLY incc='include :f7'
define LITERALLY lst = 'list :f7'
else
if device=='8' then
drive=8
define LITERALLY incc='include :f8'
define LITERALLY lst = 'list :f8'
else
if device=='9' then
drive=9
define LITERALLY incc='include :f9'
define LITERALLY lst = 'list :f9'
end;end;end;end;end;end;end;end;end;endif
flag = true
define LITERALLY i82530 = 'incc:main.inc nolist'
end
incc:main.ov0 nolist
main /*execute the procedure*/
incc:supjob.inc nolist



                                    *MAIN.OV0*
                                    **********

cury = 20t
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC main = DO
define CHAR bell = 07h
define CHAR chr

```

280722-21

```
define PROC w_com = DO
curhome;cleareos
if %0 == 'e' then
curhome;cleareos
write concat(1bh,'L',90h),'SCC Procedures can be restarted by typing "i82530"'
write concat(1bh,'L',80h),' '
lst:supjob.inc
nolist
lst:job.inc
nolist
else
lst:supjob.inc
write using ('"incc:",0,".ov0 nolist"') %0
write using ('"incc:",0,".inc nolist"') %0
curhome;cleareos
nolist
end
end

curhome;cleareos
write ' '
write concat(1bh,'L',81h),'                          *****************************************'
write concat(1bh,'L',81h),'                          *                82530/82530-6          *'
write concat(1bh,'L',81h),'                          *                                       *'
write concat(1bh,'L',81h),'                          *SERIAL COMMUNICATIONS CONTROLLER(SCC)*'
write concat(1bh,'L',81h),'                          *****************************************'
write concat(1bh,'L',80h), ' '
write ' '
write '                   1.  PIN DESCRIPTIONS            [DESCRIPTION]'
write '                   2.  REGISTER FUNCTIONS          [DESCRIPTION]'
write '                   3.  TIME CONSTANT VALUES        [DESCRIPTION]'
write '                   4.  DATA ENCODING METHODS       [DESCRIPTION]'
write '                   5.  REGISTER ADDRESSING         [DESCRIPTION]'
write '                   6.  SCC PROTOCOLS               [DESCRIPTION]'
write '                   7.  REGISTER BIT FUNCTIONS   [INTERROGATE 82530]'
write '                   8.  EXIT TO I2ICE MAIN MENU   '
write ' '
write ' '
write concat(1bh,'L',81h),'                              ----------------------'
write concat(1bh,'L',83h),'                              PLEASE TYPE YOUR CHOICE'
write concat(1bh,'L',81h),'                              ----------------------'
write concat(1bh,'L',80h),' '
repeat
chr = ci
if chr=='1' then
write using ('0') chr
w_com('pindes');return
else
if chr == '2' then
write using ('0') chr
w_com('regdes');return
else
if chr == '3' then
write using ('0') chr
w_com('timval');return
else
if chr == '4' then
write using ('0') chr
w_com('datenc');return
else
if chr == '5' then
write using ('0') chr
w_com('regadd');return
else
if chr== '6' then
write using ('0') chr
w_com('prtcol');return
else
if chr == '7' then
write using ('0') chr
w_com('pinfun');return
else
if chr == '8' then
write using ('0') chr
w_com('e');return
else
bell
end
end
end
end
end
```

280722-22

```
end
end
end
end
end



                                   *MAIN.INC*
                                   **********

main
incc:supjob.inc nolist



                                   *PINDES.OVO*
                                   ************

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define CHAR ppp
define PROC pinds = DO
curhome;cleareos
write concat(1bh,'L',81h),'*82530/82530-6 SERIAL COMMUNICATIONS CONTROLLER (SCC)*'
write concat(1bh,'L',81h),'******************************************************** '
write concat(1bh,'L',81h),'                   PIN DESCRIPTION'
write concat(1bh,'L',81h),'                   ***************'
write concat(1bh,'L',81h),' SYMBOL   PIN NO.   TYPE    NAME AND FUNCTION'
write concat(1bh,'L',81h),' @@@@@@   @@@@@@@   @@@@    @@@@@@@@@@@@@@@@@'
write concat(1bh,'L',80h),' '
end
define PROC pindes = DO
pinds
write '     DB0       40      I/O     DATA BUS:The Data Bus lines are '
write '     DB1        1      I/O     bi-directional three-state lines'
write '     DB2       39      I/O     which interface with the systems'
write '     DB3        2      I/O     Data Bus. These lines carry data'
write '     DB4       38      I/O     and commands to and from the SCC.'
write '     DB5        3      I/O'
write '     DB6       37      I/O'
write '     DB7        4      I/O'
write ' '
write '     INT        5       O      INTERRUPT REQUEST:The interrupt signal'
write '                               is activated when the SCC requests an'
write '                               interrupt. It is an open drain output.'
write concat(1bh,'1',83h),'               PLEASE HIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
ppp = ci
pinds
write '     IEO        6       O      INTERRUPT ENABLE OUT:is high only if IEI'
write '                               is High and the CPU is not servicing an'
write '                               SCC interrupt or the SCC is not requesting'
write '                               an interrupt.'
write ' '
write '     IEI        7       I      INTERRUPT ENABLE IN:is used with IEO to form'
write '                               an interrupt daisy chain when there is more'
write '                               than one interrupt-driven device.'
write '    ____'
write '    INTA        8       I      INTERRUPT ACKNOWLEDGE:indicates an active '
write '                               interrupt acknowledge cycle. During this '
write '                               cycle, the SCC interrupt daisy chain settles.'
write concat(1bh,'L',83h),'                PLEASE HIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
ppp = ci
pinds
write '     VCC        9              POWER:5V Power supply'
write '    ___ ___'
write '    RDYa/REQa  10       O      READY/REQUEST:(Output, open-drain when '
write '    ___ ___                    programmed for a Ready function, driven'
write '    RDYb/REQb           O      high or Low when programmed for a Request func.'
write '    ____'
write '    SYNCa      11      I/O     SYNCHRONIZATION: These pins can attract'
write '                               either as inputs, outputs or part of the'
write '                               crystal oscillator circuit.'
write '    ____'
write '    RTxCa      12       O      RECEIVE/TRANSMIT CLOCKS: These pins can be'
write '    ____                       be programmed in several different modes of'
write '    RTxCb                      operation.  '
write concat(1bh,'L',83h),'                PLEASE HIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
```

280722-23

```
ppp = ci
pinds
write '     ____'
write '     TRxCa       14        I/O      TRANSMIT/RECEIVE CLOCKS:These pins can be'
write '     ____                           programmed in several different modes of'
write '     TRxCb       26        I/O      operation. The receive clock or the transmit'
write '                                    clock in the input mode or the output of'
write '                                    the Digital Phase Locked Loop, the crystal'
write '                                    oscillator, the baud rate generator, or the'
write '                                    transmit clock in the output mode may be'
write '                                    supplied.'
write concat(lbh,'l',83h),'                        PLEASE HIT ANY KEY TO CONTINUE'
write concat(lbh,'L',80h),''
ppp = ci
end
define PROC pndsad  = DO
pinds
write '      TxDa       15         O       TRANSMIT DATA:These output signals transmit'
write '     .TxDa       25         O       serial data at standard TTL levels.'
write '     ___ ___'
write '    DTRaREQa     16         O       DATA TERMINAL READY/REQUEST:These outputs'
write '    ___ ___                         follow the state programmed into the DTR bit'
write '    DTRbREQb     24         O       They can also be used as general purpose'
write '                                    outputs or as Request lines for a DMA contr.'
write '     ___'
write '     RTSa        17         O       REQUEST TO SEND:When the RTS bit in write'
write '     ___                            Register 5 is set, the signal goes low. '
write '     RTSb        23         O       When the RTS bit is reset in the Async mode'
write '                                    and Auto enable is on, the signal goes high'
write '                                    after the transmitter is empty.'
write concat(lbh,'l',83h),'                        PLEASE HIT ANY KEY TO CONTINUE'
write concat(lbh,'L',80h),''
ppp = ci
pinds
write '      RxDa       13         I       RECEIVE DATA:These lines receive serial'
write '      RxDb       27         I       data at standard TTL Levels'
write '     ___'
write '     CTSa        18         I       CLEAR TO SEND:If these pins are programmed'
write '     ___                            as Auto Enables, a Low on the inputs enables'
write '     CTSb        22         I       the respevtive transmitters. If not'
write '                                    programmed as Auto Enables, they may be used'
write '                                    as general purpose inputs.'
write '     __'
write '     CDa         19         I       CARRIER DETECT:These pins function as'
write '     __                             receiver enables if they are programmed'
write '     CDb         21         I       for Auto Enables; otherwise they may be used'
write '                                    as general purpose input pins.'
write concat(lbh,'L',83h),'                        PLEASE HIT ANY KEY TO CONTINUE'
write concat(lbh,'L',80h),''
ppp = ci
pinds
write '      CLK        20         I       CLOCK:This is the system SCC clock used to'
write '                                    synchronize internal signals.(TTL level)'
write '       _'
write '      D/C        32         I       DATA/COMMAND SELECT:This signal defines the'
write '                                    type of information transferred to or from'
write '                                    the SCC. High means data is transferred, a'
write '                                    Low indicates a command.'
write '      __'
write '      CS         33         I       CHIP SELECT:This signal selects the SCC for '
write '                                    a read or write operation.'
write concat(lbh,'L',83h),'                        PLEASE HIT ANY KEY TO CONTINUE'
write concat(lbh,'L',80h),''
ppp = ci
pinds
write '        _'
write '      A/B        34         I       CHANNEL A/CHANNEL B SELECT:This signal '
write '                                    selects the channel in which the read or'
write '                                    write operation occurs.'
write '       __'
write '       WR        35         I       WRITE:When the SCC is selected this signal'
write '                                    indicates a write operation.'
write '       __'
write '       RD        36         I       READ:This signal indicates a read operation and'
write '                                    when the SCC is selected, enables its bus drivers.'
write ' '
write '      GND        31                 GROUND'
write concat(lbh,'l',83h),'                 PLEASE HIT ANY KEY TO RETURN TO SCC MENU'
write concat(lbh,'L',80h), ' '
ppp = ci
lst:job.inc
write 'remove ppp'
```

280722-24

```
write 'remove pindes,pndsad,pinds'
write 'i82530'
nolist
end


                                    *PINDES.INC*
                                    ************

pindes
pndsad
incc:job.inc nolist



                                    *REGDES.OVO*
                                    ************

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC regdes = DO
repeat;curhome;cleareos
write concat(1bh,'L',81h),'                ***************************************'
write concat(1bh,'L',81h),'                *READ AND WRITE REGISTER DESCRIPTION*'
write concat(1bh,'L',81h),'                ***************************************'
write concat(1bh,'L',80h),' '
write '                        1. READ REGISTER DESCRIPTION'
write '                        2. WRITE REGISTER DESCRIPTION'
write '                        3. EXIT TO SCC MAIN MENU'
write ' '
write concat(1bh,'L',83h),'                      PLEASE TYPE YOUR CHOICE'
write concat(1bh,'L',80h),' '
define CHAR yyy = ci
until yyy == '3'
if yyy =='1' then
curhome;cleareos
write concat(1bh,'L',81h),'READ REGISTER DESCRIPTION          '
write concat(1bh,'L',80h), ' '
write 'RR0   Transmit/Receive buffer status and External status'
write ' '
write 'RR1   Special Receive Condition status'
write ' '
write 'RR2   Modified Interrupt vector (Channel B only)'
write '      Unmodified Interrupt (Channel A only)'
write ' '
write 'RR3   Interrupt Pending bits (Channel A only)'
write ' '
write 'RR8   Receive buffer'
write ' '
write 'RR10  Miscellaneous status'
write ' '
write 'RR12  Lower byte of baud rate generator time constant'
write ' '
write 'RR13  Upper byte of baud rate generator time constant'
write ' '
write 'RR15  External/Status interrupt information'
write concat(1bh,'L',83h),'                     PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'L',80h),''
define CHAR zzz = ci
else
if yyy == '2' then
curhome;cleareos
write concat(1bh,'L',81h),'WRITE REGISTER DESCRIPTION'
write concat(1bh,'L',80h),' '
write 'WR0   CRC initialize, initialization commands for the various modes,'
write '      shift right/shift left command.'
write ' '
write 'WR1   Transmit/Receive interrupt and data transfer mode definition'
write ' '
write 'WR2   Interrupt vector (accessed through either channel)'
write ' '
write 'WR3   Receive parameters and control'
write ' '
write 'WR4   Transmit/Receive miscellaneous parameters and modes'
write ' '
write 'WR5   Transmit parameters and controls'
write ''
write 'WR6   Sync characters or SDLC address field'
write concat(1bh,'L',83h),' PLEASE HIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
define CHAR zzz = ci
```

280722-25

```
curhome;cleareos
write 'WR7   Sync characters or SDLC address field'
write ''
write 'WR8   Transmit buffer'
write ' '
write 'WR9   Master interrupt control and reset (accessed through either'
write '      channel)'
write ' '
write 'WR10  Miscellaneous transmitter/receiver control bits'
write ' '
write 'WR11  Clock mode control'
write ' '
write 'WR12  Lower Byte of baud rate generator time constant'
write ' '
write 'WR13  Upper Byte of baud rate generator time constant'
write ' '
write 'WR14  Miscellaneous control bits'
write ' '
write 'WR15  External/Status interrupt control'
write concat(1bh,'L',83h),'                          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'L',80h), ' '
define CHAR zzz = ci
end
end
end
lst:job.inc
write 'remove regdes'
write 'i82530'
nolist
end



                                 *REGDES.INC*
                                 ************

regdes
incc:job.inc nolist



                                 *TIMVAL.OVO*
                                 ************

define PROC timval = DO
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
curhome;cleareos
write concat(1bh,'L',81h),'TIME CONSTANT VALUES FOR STANDARD BAUD RATES
write concat(1bh,'L',81h),'********************************************
write concat(1bh,'L',80h), ' '
write '            BAUD RATE    TIME CONSTANT    ERROR'
write '              19200          102           -'
write '               9600          206           -'
write '               7200          275           0.12%'
write '               4800          414           -'
write '               3600          553           0.06%'
write '               2400          830           -'
write '               2000          996           0.04%'
write '               1800         1107           0.03%'
write '               1200         1662           -'
write '                600         3326           -'
write '                300         6654           -'
write '                150        13310           -'
write '                134.5      14844           0.0007%'
write '                110        18151           0.0015%'
write '                 75        26622           -'
write '                 50        39934           -'
write concat(1bh,'L',83h),'              PLEASE HIT ANY KEY TO RETURN TO SCC MENU'
write concat(1bh,'L',80h), ' '
define CHAR yyy = ci
lst:job.inc
write 'remove timval'
write 'i82530'
nolist
curhome;cleareos;end



                                 *TIMVAL.INC*
                                 ************

```

280722-26

```
timval
incc:job.inc nolist



                               *DATENC.OV0*
                               ************

curhome;cleareos
write concat(lbh,'L',83h),'LOADING....'
write concat(lbh,'L',80h),''
define PROC dateno = DO
repeat
curhome;cleareos
write concat(lbh,'L',81h),'                     ***********************'
write concat(lbh,'L',81h),'                     *DATA ENCODING METHODS* '
write concat(lbh,'L',81h),'                     ***********************'
write concat(lbh,'L',80h), ' '
write ' '
write ' '
write '                        1. DATA TABLE '
write '                        2. TIMING DIAGRAM'
write '                        3. EXIT TO SCC MAIN MENU'
write ' '
write ' '
write ' '
write ' '
write ' '
write ' '
write ' '
write concat(lbh,'L',83h),'                     PLEASE TYPE YOUR CHOICE'
write concat(lbh,'L',80h),''
define CHAR zzz = ci
until zzz=='3'
if zzz=='1' then
curhome;cleareos
write concat(lbh,'L',81h),'MODE         SYSTEM      SYSTEM CLOCK    SERIAL      CONDITIONS'
write concat(lbh,'L',81h),'             CLOCK       SERIAL CLOCK    BIT RATE  '
write concat(lbh,'L',81h),'****         ******      ************    ********    **********'
write concat(lbh,'L',80h), ' '
write 'Serial       4MHz             4          1 Mbps      Serial clocks'
write 'clocks                                               synchronized with'
write 'generated                                            system clock'
write 'externally   6MHz             4          1.5 Mbps    Same as above'
write ' '
write '             4MHz             4.5        880 Kbps    Serial clocks and'
write '                                                     system clock async'
write '             6MHz             4.5        1.3 Mbps    Same as above'
write ''
write 'Self-clocked '
write 'Operation'
write 'NRZI         4MHz             32         125 Kbps'
write '             6MHz             32         187 Kbps'
write ' '
write 'FM           4MHz             16         250 Kbps'
write '             6MHz             16         375 Kbps'
write concat(lbh,'l',83h), '                         PLEASE HIT ANY KEY TO RETURN'
write concat(lbh,'L',80h),''
define CHAR yyy=ci
else if zzz=='2' then
curhome;cleareos
write concat(lbh,'L',81h),'                       TIMING DIAGRAM '
write concat(lbh,'L',81h),'                       ************** '
write concat(lbh,'L',80h), ' '
write ' DATA    1    |   1   |   0    |   0   |  1   |   0   '
write '      _________________                _________       HIGH = 1'
write '  NRZ                 |                |      |       '
write '                     --------------------     ---------LOW = 0'
write '  NRZ1_________________        _________________       NO CHANGE=1'
write '                      |        |              |'
write '                     -----------              ---------CHANGE = 0'
write '  FM1     _____  ______        __________ ______       BIT CENTER'
write '     |   |    |   |   |        |       |  |   |       |TRANSITION'
write '     -----   -----   -----------      ----    ---------TRANSITION=1'
write '                                                       NO TRANSITION=0'
write ' FM0         __________  _______   ______     ______   NO TRANSITION=1'
write '     |        |       |   |    |   |   |      |   |   |TRANSITION=0'
write '     ----------      ------    -----    --------  -----'
write 'MAN-  _____  ______      _______   _________      ______ LOW-HIGH=0'
write 'CHE-     |    |   |       |    |   |      |       |      HIGH-LOW=1'
write 'STER     ------  ----------    -----    ------------'
write concat(lbh,'l',83h), '                PLEASE HIT ANY KEY TO RETURN'
```

280722-27

```
write concat(lbh,'L',80h),''
define CHAR yyy=ci
end
end
end
lst:job.inc
write 'remove datenc'
write '182530'
nolist
curhome;cleareos;end



                                *DATENC.INC*
                                ************

datenc
incc:job.inc nolist



                                *REGADD.OVO*
                                ************

curhome ; cleareos
write concat(lbh,'L',83h),'LOADING....'
write concat(lbh,'L',90h),''
define PROC regadd = DO
curhome ; cleareos
write concat(lbh,'L',81h),'D/C "POINT HIGH" CODE     D2    D1     D0      WRITE        READ'
write concat(lbh,'L',81h),'          IN WRO               IN WRO           REGISTER    REGISTER'
write concat(lbh,'L',80h), ' '
write '  high        Either way    X      X      X      Data        Data'
write '  Low         Not true      0      0      0       0           0'
write '  Low         Not true      0      0      1       1           1'
write '  Low         Not true      0      1      0       2           2'
write '  Low         Not true      0      1      1       3           3'
write '  Low         Not true      1      0      0       4          (0)'
write '  Low         Not true      1      0      1       5          (1)'
write '  Low         Not true      1      1      0       6          (2)'
write '  Low         True          1      1      1       7          (3)'
write '  Low         True          0      0      0      Data        Data'
write '  Low         True          0      0      1       9           -'
write '  Low         True          0      1      0      10          10'
write '  Low         True          0      1      1      11          (15)'
write '  Low         True          1      0      0      12          12'
write '  Low         True          1      0      1      13          13 '
write '  Low         True          1      1      0      14          (10)'
write '  Low         True          1      1      1      15          15'
write concat(lbh,'L',83h),'            PLEASE HIT ANY KEY TO RETURN TO SCC MENU'
write concat(lbh,'L',80h), ' '
define CHAR yyy = ci
lst:job.inc
write 'remove regadd'
write '182530'
nolist
curhome;cleareos
end



                                *REGADD.INC*
                                ************

regadd
incc:job.inc nolist



                                *PRTCOL.OVO*
                                ************

curhome;cleareos
write concat(lbh,'L',83h),'LOADING....'
write concat(lbh,'L',80h),''
define PROC prtcol= DO
curhome;cleareos
write concat(lbh,'L',81h),'                                   SCC PROTOCOLS'
write concat(lbh,'L',80h),'         Start   Parity'
write '_______  _____________  _____________  _________________'
write 'Marking¶  ¶Data ¶ ¶  '  ¶  ¶Data ¶ ¶  '  ¶  ¶Data ¶  ¶ ' ' Marking '
write 'Line     -----------     -----------     -----------      Line'
```

280722-28

```
write '                             "ASYNCaRONOUS"'
write '  __________________________/ /___________________________________'
write '  ¶SYNC    ¶ DATA  ¶      "MONOSYNC"        ¶ DATA    ¶  CRC1  ¶CRC2 ¶'
write '  --------------------------/ /-----------------------------------'
write '____________________________/ /___________________________________'
write '¶ ¶ SYNC   ¶ DATA  ¶        "BISYNC"        ¶ DATA    ¶  CRC1  ¶CRC2 ¶'
write '----------------------------/ /-----------------------------------'
write '           Signal          '
write '            ________________/ /___________________________________'
write '            ¶ DATA  ¶     "EXTERNAL SYNC"   ¶ DATA    ¶  CRC1  ¶CRC2 ¶'
write '            ----------------/ /-----------------------------------'
write '____________________________/ /_________________________________________'
write '¶ ¶ADDR    ¶                  INFORMATION             ¶ CRC1   ¶CRC2 ¶     ¶'
write '----------------------------/ /-----------------------------------------'
write 'FLAG                       "SDLC/HDLC/X.25"'
write concat(1bh,'L',83h),'                   PLEASE nIT ANY KEY TO RETURN TO SCC MENU'
wmite concat(1bh,'L',80h),''
define CnAR yyy = ci
lst:job.inc
write 'remove prtcol'
write '182530'
nolist
curhome;cleareos
end



                            "PRTCOL.INC"
                            ************

prtcol
incc:job.inc nolist



                            "PINFUN.OVO"
                            ************

curhome;cleareos
write concat(1bh,'L',83h),'LOADING ....'
write concat(1bh,'L',80h), ' '
curhome;cleareos

define WORD paddr
define WORD paddr1
define CHAR chrin
define BOOLEAN check = true
define WORD wreg
define BYTE regnum
define CHAR bell = 07h

define PROC gethex = DO
define WORD num
define CHAR chr
num = 0
repeat
chr = ci
if (chr >= '0') and (chr <='9') then
num = num*10h + (chr-30h)
write using ('1,>') chr
else
if (chr >= 'A') and (chr <= 'F') then
num = num*10h + (chr-37h)
write using ('1,>') chr
else
if (chr >= 'a') and (chr <= 'f') then
num = num*10h + (chr-57h)
write using ('1,>') chr
else
if chr <> 0dh then write using('0,>') bell
end
end
end
end
until chr == 0dh
end
return num
end

define PROC getnum = DO
define BYTE num
define CnAR chr
```

280722-29

```
define CHAR bell = 07h
num = 0
repeat
chr = ci
if (chr >= '0') and (chr <='9') then
num = num*10t + (chr-30h)
write using ('1,>') chr
else
if chr <> 0dh then write using('0,>') bell
end
end
until chr == 0dh
end
return num
end

define PROC getpl = DO
curhome;cleareos
define CnAR zzz
write concat(1bh,'L',81h),'MAPIO COMMAND DISPLAYS OR SETS PHYSICAL LOCATION FOR I/O PORTS'
write concat(1bh,'L',83h),' '
write using('"              nAVE YOU MAPPED TnE I/O PORTS?(Y/N)",>') chrin
write concat(1bh,'L',80h),' '
repeat
chrin = ci
if not((chrin == 'N') or (chrin == 'n') or (chrin == 'Y') or (chrin == 'y')) then
bell
endif
until (chrin == 'N') or (chrin == 'n') or (chrin == 'Y') or (chrin == 'y')
end
if (chrin == 'N') or (chrin == 'n') then do
write using('0') chrin
curhome;cleareos
write '           TnE I/O PORTS CAN BE MAPPED TnE FOLLOWING WAY'
write ' '
write '                MAPIO [(partition) <USER or ICE>]'
write ''
write '    MAPIO - displays the current map of I/O port address blocks'
write ''
write ' partition - is an entry specifying a range of addresses such as:'
write '             Starting port-address LENGTn number of bytes'
write ''
write '     USER - Maps I/O to the USER system'
write ''
write '      ICE - Maps I/O to the I2ICE probe'
write ''
write concat(1bh,'L',83h),' PLEASE TYPE MAPIO STARTING nEXADECIMAL PORT-ADDRESS and <cr>'
write concat(1bh,'L',80h), ' '
paddr = gethex
write 'n'
write concat(1bh,'L',83h),'               PLEASE TYPE NUMBER OF BYTES and <cr>'
write concat(1bh,'L',80h), ' '
paddrl = gethex
write 'n'
write concat(1bh,'L',83h),'            PLEASE TYPE ''U'' FOR USER or ''I'' FOR ICE'
write concat(1bh,'L',80h),''
repeat
zzz = ci
if not((zzz == 'U') or (zzz == 'u') or (zzz == 'I') or (zzz == 'i')) then
bell
endif
until (zzz == 'U') or (zzz == 'u') or (zzz == 'I') or (zzz == 'i')
end
if (zzz=='I') or (zzz=='i') then
MAPIO PADDR LENGTH PADDRl ICE
write using('"MAPIO ",0,n," LENGTn ",0,n," ICE "')paddr,paddrl
write ''
write concat(1bh,'L',83h),'                    PLEASE HIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
zzz = ci;else
if (zzz=='U') or (zzz=='u') then
MAPIO PADDR LENGTH PADDRl USER
write using('"MAPIO ",0,n," LENGTn ",0,n," USER "')paddr,paddrl
write ''
write concat(1bh,'L',83h),'                    PLEASE nIT ANY KEY TO CONTINUE'
write concat(1bh,'L',80h),''
chrin = ci;endif;end
end
end
curhome;cleareos
write concat(1bh,'L',83h),'     TYPE IN THE HEXADECIMAL ADDRESS FOR THE 82530 and <cr>'
write concat(1bh,'L',80h),' '
```

280722-30

```
wreg = gethex
write 'n'
write ''
end

define PROC getp2 = DO
curhome;cleareos
write concat(1bh,'L',81h),'                                   ********************'
write concat(1bh,'L',81h),'                                   *PIN FUNCTION MENU**'
write concat(1bh,'L',81h),'                                   ********************'
write concat(1bh,'L',80h),' '
write ' '
write ' '
write '                       "R"  -  READ A REGISTER'
write '                       "W"  -  WRITE TO A REGISTER'
write '                       "N"  -  CHANGE 82530 PORT ADDRESS'
write '                       "E"  -  EXIT TO MAIN MENU'
write ' '
write ' '
write concat(1bh,'L',83h),'                                PLEASE TYPE YOUR CHOICE'
write concat(1bh,'L',80h),' '
end

define PROC getp3 = DO
curhome;cleareos
write ' '
write concat(1bh,'L',81h),'    ************************************************ '
write concat(1bh,'L',81h),'    THE CHOICES OF REGISTERS FOR YOUR SELECTION ARE :'
write concat(1bh,'L',81h),'    ************************************************'
write concat(1bh,'L',80h), ' '
if (chrin == 'R') or (chrin == 'r') then
write '     0. RR0               1. RR1               2. RR2'
write '     3. RR3               8. RR8              10. RR10'
write '    12. RR12             13. RR13             15. RR15'
else
write '     0. WR0               1. WR1               2. WR2'
write '     3. WR3               4. WR4               5. WR5'
write '     6. WR6               7. WR7               8. WR8'
write '     9. WR9              10. WR10             11. WR11'
write '    12. WR12             13. WR13             14. WR14'
write '                         15. WR15'
end
write ' '
write ' '
write concat(1bh,'L',83h),'                        PLEASE TYPE YOUR CHOICE and <cr>'
write concat(1bh,'L',80h),' '
regnum = getnum
write 'T'
write ''
end

define PROC getmenl = DO
if check then
getpl
check = false
end
repeat
getp2
chrin = ci
if (chrin == 'E') or (chrin == 'e') then write using ('0') chrin;return end
if (chrin == 'R') or (chrin == 'r') or (chrin == 'W') or (chrin == 'w') then
write using ('0') chrin
curhome;cleareos
getp3
return
else
if (chrin == 'N') or (chrin == 'n') then
write using ('0') chrin
getpl
else
if (chrin == 1bh) then return end
bell
end
end
end
end

define PROC w_coml = DO
curhome;cleareos
if %0 == 'e' then
lst:job.inc
write 'remove done,gethex,getnum,getmenl,w_coml,getpl,getp2,getp3'
```

280722-31

```
write 'remove chrin,check,wreg,regnum,paddr,paddrl'
write '182530'
nolist
else
lst:job.inc
write using ('"incc:",0," nolist"') %0
write using ('0') %0
write using ('"remove ",0') %0
write 'incc:pinfun.inc nolist'
nolist
end
end

define PROC done = DO
curhome ; cleareos
getmenl
if (chrin == 'E') or (chrin == 'e') then
w_coml('e')
return
end
repeat
if (chrin == 'R') or (chrin == 'r') then
do
if regnum == 0 then
w_coml('rr0') ; return
end
if regnum == 1t then
w_coml('rr1');return
end
if regnum == 2t then
w_coml('rr2'); return
end
if regnum == 3t then
w_coml('rr3'); return
end
if regnum == 8t then
w_coml('rr8'); return
end
if regnum == 10t then
w_coml('rr10'); return
end
if regnum == 12t then
w_coml('rr12'); return
end
if regnum == 13t then
w_coml('rr13'); return
end
if regnum == 15t then
w_coml('rr15'); return
end
end
end
if (chrin == 'R') or (chrin == 'r')  then
write using ('"UNABLE TO READ REGISTER ",0,>') regnum
getp3
end
if (chrin == 'W') or (chrin == 'w') then
do
if regnum == 0 then
w_coml('wr0'); return
end
if regnum == 1t then
w_coml('wr1'); return
end
if regnum == 2t then
w_coml('wr2'); return
end
if regnum == 3t then
w_coml('wr3'); return
end
if regnum == 4t then
w_coml('wr4'); return
end
if regnum == 5t then
w_coml('wr5'); return
end
if regnum == 6t then
w_coml('wr6'); return
end
if regnum == 7t then
w_coml('wr7'); return
end
```

280722–32

```
if regnum == 8t then
w_com1('wr8'); return
end
if regnum == 9t then
w_com1('wr9'); return
end
if regnum == 10t then
w_com1('wr10'); return
end
if regnum == 11t then
w_com1('wr11'); return
end
if regnum == 12t then
w_com1('wr12'); return
end
if regnum == 13t then
w_com1('wr13'); return
end
if regnum == 14t then
w_com1('wr14'); return
end
if regnum == 15t then
w_com1('wr15'); return
end
end
end
if (chrin == 'W') or (chrin == 'w') then
write using ('"UNABLE TO WRITE TO REGISTER ",0,>') regnum
getp3
end
end
end



                                 *PINFUN.INC*
                                 ************


done
incc:job.inc nolist



                                 *RR0*
                                 *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr0 = DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome ;cleareos
write concat(1bh,'L',81h),'                          READ REGISTER 0'
write concat(1bh,'L',81h),'                          ***************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'_____________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-----------------------------------------------------'
write '    D7  =   BREAK/ABORT'
write '          D6  =   Tx UNDERRUN/EOM'
write '                D5  =   CTS'
write '                      D4  =   SYNC/HUNT'
write '                            D3  =   CD'
write '                                  D2  =   Tx BUFFER EMPTY'
write '                                        D1  =   ZERO COUNT'
write '                                              D0  =   Rx CHARACTER AVAILABLE'
write ' '
write concat(1Bh,'L',83h),'               PLEASE HIT ANY KEY TO RETURN       '
write concat(1bh,'L',80h), ' '
yyy=ci
end
```

280722-33

```



                                        *RR1*
                                        *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr1 = DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                                   READ REGISTER 1'
write concat(1bh,'L',81h),'                                   ***************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'_______________________________________________________________'
write concat(1bh,'L',80h),'¶ D7   ¶ D6   ¶ D5   ¶ D4   ¶ D3   ¶ D2   ¶ D1   ¶ D0   ¶'
write concat(1bh,'L',80h),'---------------------------------------------------------------'
write '    D7  =    END OF FRAME'
write '              D6  =    CRC/FRAMING ERROR'
write '                         D5  =    Rx OVERRUN ERROR'
write '                                    D4  =    PARITY ERROR'
write '                                               D3  =    RESIDUE CODE 0'
write '                                                          D2  =    RESIDUE CODE 1'
write '                                                                     D1  =    RESIDUE CODE 2'
write '                                                                                D0  =    ALL SENT'
write ' '
write concat(1bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
temp = ci
end



                                        *RR2*
                                        *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr2 = DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                                   READ REGISTER 2'
write concat(1bh,'L',81h),'                                   ***************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'_______________________________________________________________'
write concat(1bh,'L',80h),'¶ D7   ¶ D6   ¶ D5   ¶ D4   ¶ D3   ¶ D2   ¶ D1   ¶ D0   ¶'
write concat(1bh,'L',80h),'---------------------------------------------------------------'
write '    D7  =    INTERRUPT VECTOR V7'
write '              D6  =    INTERRUPT VECTOR V6'
write '                         D5  =    INTERRUPT VECTOR V5'
write '                                    D4  =    INTERRUPT VECTOR V4'
write '                                               D3  =    INTERRUPT VECTOR V3'
write '                                                          D2  =    INTERRUPT VECTOR V2'
write '                                                                     D1  =    INTERRUPT VECTOR V1'
write '                                                                                D0  =    INTERRUPT VECTOR V0'
write ' '
```

280722–34

```
write 'NOTE: Interrupt Vector is modified in B Channel'
write ' '
write concat(1Bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
temp=ci
end



                                   *RR3*
                                   *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr3 = DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                          READ REGISTER 3'
write concat(1bh,'L',81h),'                          ***************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'------------------------------------------------'
write '    D7  =   0'
write '          D6  =   0'
write '                D5  =   CHANNEL A Rx IP'
write '                      D4  =   CHANNEL A Tx IP'
write '                            D3  =   CHANNEL A EXT/STAT IP'
write '                                  D2  =   CHANNEL B Rx IP'
write '                                        D1  =   CHANNEL B Tx IP'
write '                                              D0  =   CHANNEL B EXT/STAT IP'
write ' '
write 'NOTE: Always in B Channel'
write ' '
write concat(1BH,'L',83H),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
temp=ci
end



                                   *RR8*
                                   *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr3 = DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                          READ REGISTER 8'
write concat(1bh,'L',81h),'                          ***************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'------------------------------------------------'
write ''
```

280722-35

```
write ''
write ''
write ''
write ''
write ''
write ''
write ''
write ' '
write ''
write ' '
write concat(1bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'L',80h), ' '
temp=ci
end



                                        'RR10'
                                        ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr10= DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                         READ REGISTER 10'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'_______________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-----------------------------------------------'
write '    D7  =   ONE CLOCK MISSING'
write '             D6  =   TWO CLOCKS MISSING'
write '                      D5  =   0'
write '                               D4  =   LOOP SENDING'
write '                                        D3  =   0'
write '                                                 D2  =   0'
write '                                                          D1  =   ON LOOP'
write '                                                                   D0  =   0'
write ' '
write concat(1bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'L',80h), ' '
temp=ci
end



                                        'RR12'
                                        ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr12= DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                         READ REGISTER 12'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),' '
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
```

280722-36

```
(temp and 01h) /01h
write concat(1bh,'L',80h),'_________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------'
write '     D7  =   TC7'
write '               D6  =   TC6'
write '                          D5  =   TC5'
write '                                    D4  =   TC4'
write '                                              D3  =   TC3'
write '                                                        D2  =   TC2'
write '                                                                  D1  =   TC1'
write '                                                                            D0  =   TC0'
write ' '
write 'NOTE: Lower Byte of TIME CONSTANT'
write ' '
write concat(1Bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
temp=ci
end



                                    *RR13*
                                    ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr13= DO
define CHAR yyy
define BYTE temp
temp  = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                   READ REGISTER 13'
write concat(1bh,'L',81h),'                   ****************'
write concat(1bh,'L',80h),' '
write using ('2o,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'_________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------'
write '     D7  =   TC15'
write '               D6  =   TC14'
write '                          D5  =   TC13'
write '                                    D4  =   TC12'
write '                                              D3  =   TC11'
write '                                                        D2  =   TC10'
write '                                                                  D1  =   TC9'
write '                                                                            D0  =   TC8'
write ' '
write 'NOTE: Upper Byte of TIME CONSTANT'
write ' '
write concat(1BH,'L',83H),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
temp=ci
end



                                    *RR15*
                                    ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC rr15= DO
define CHAR yyy
define BYTE temp
temp = port(wreg)
curhome
cleareos
write concat(1bh,'L',81h),'                   READ REGISTER 15'
write concat(1bh,'L',81h),'                   ****************'
write concat(1bh,'L',80h),' '
```

280722-37

```
write using ('2c,"=>",2,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y,5x,1,y') &
(temp and 80h) /80h, &
(temp and 40h) /40h, &
(temp and 20h) /20h, &
(temp and 10h) /10h, &
(temp and 08h) /08h, &
(temp and 04h) /04h, &
(temp and 02h) /02h, &
(temp and 01h) /01h
write concat(1bh,'L',80h),'______________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'--------------------------------------------------------------'
write '     D7  =   BREAK/ABORT IE'
write '              D6  =   Tx UNDERRUN/EOM IE'
write '                     D5  =   CTS IE'
write '                            D4  =   SYNC/HUNT IE'
write '                                   D3  =   CD IE'
write '                                          D2  =   0'
write '                                                 D1  =   ZERO COUNT IE'
write '                                                        D0  =   0'
write ' '
write concat(1BH,'L',83H),'         PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
temp=ci
end



                                  *WR0*
                                  *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr0 = DO
define CHAR yyy
define BYTE temp
curhome;cleareos
write concat(1bh,'L',81h),'                      WRITE REGISTER 0'
write concat(1bh,'L',81h),'                      ****************'
write concat(1bh,'L',80h),'______________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'--------------------------------------------------------------'
write '                         D7 D6'
write '                         0  0  =  NULL CODE '
write '                         0  1  =  RESET Rx CRC CHECKER'
write '                         1  0  =  RESET Tx CRC GENERATOR'
write '                         1  1  =  RESET Tx UNDERRUN/EOM LATCH'
write ' D5 D4 D3                                       D2 D1 D0'
write ' 0  0  0  =  NULL CODE                          0  0  0  =  0 or  8'
write ' 0  0  1  =  POINT HIGH REGISTER GROUP          0  0  1  =  1 or  9'
write ' 0  1  0  =  PRESET EXT/STATUS INTERRUPTS       0  1  0  =  2 or 10'
write ' 0  1  1  =  SEND ABORT                         0  1  1  =  3 or 11'
write ' 1  0  0  =  ENABLE INT ON NEXT Rx CHAR.        1  0  0  =  4 or 12'
write ' 1  0  1  =  RESET TxINT PENDING                1  0  1  =  5 or 13'
write ' 1  1  0  =  ERROR RESET                        1  1  0  =  6 or 14'
write ' 1  1  1  =  RESET HIGHEST IUS                  1  1  1  =  7 or 15'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'n'
port(wreg)=0
port(wreg) = temp
write ' '
write concat(1Bh,'L',83h),'         PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
yyy=ci
end



                                  *WR1*
                                  *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr1 = DO
define CHAR yyy
define BYTE temp
curhome;cleareos
```

280722-38

```
write concat(1bh,'L',81h),'                              WRITE REGISTER 1'
write concat(1bh,'L',81h),'                              ****************'
write concat(1bh,'L',80h),'_______________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------'
write '    D7  =   READY/DMA REQUEST ENABLE'
write '              D6  =   READY/DMA REQUEST FUNCTION'
write '                        D5  =   READY/DMA REQUEST ON RECEIVE/TRANSMIT'
write '                                  D4    D3    =  0 0  =  Rx INT DISABLE'
write '                                                =  0 1  =  Rx INT ON FIRST CHAR OR CONDN'
write '                                                =  1 0  =  INT ON ALL Rx CHAR OR CONDN'
write '                                                =  1 1  =  Rx INT ON SPECIAL CONDN ONLY'
write '                                                  D2  =   PARITY IS SPECIAL CONDITION'
write '                                                            D1  =   Tx INT ENABLE'
write '                                                                      D0  =   EXT. INT ENABLE'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 1t
port(wreg) = temp
write concat(1BH,'L',83H),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy = ci
end



                                    *WR2*
                                    *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr2 = DO
define CHAR yyy
define BYTE temp
curhome;cleareos
write concat(1bh,'L',81h),'                              WRITE REGISTER 2'
write concat(1bh,'L',81h),'                              ****************'
write concat(1bh,'L',80h),'_______________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------'
write '    D7  =   INTERRUPT VECTOR V7'
write '              D6  =   INTERRUPT VECTOR V6'
write '                        D5  =   INTERRUPT VECTOR V5'
write '                                  D4  =   INTERRUPT VECTOR V4'
write '                                            D3  =   INTERRUPT VECTOR V3'
write '                                                      D2  =   INTERRUPT VECTOR V2'
write '                                                                D1  =   INTERRUPT VECTOR V1'
write '                                                                          D0  =   INTERRUPT VECTOR V0'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 2t
port(wreg) = temp
write concat(1BH,'L',83H),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



                                    *WR3*
                                    *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr3 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                              WRITE REGISTER 3'
write concat(1bh,'L',81h),'                              ****************'
write concat(1bh,'L',80h),'_______________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------'
write '    D7      D6  =  0 0  =  Rx 5 BITS/CHARACTER'
write '                =  0 1  =  Rx 7 BITS CHARACTER'
write '                =  1 0  =  Rx 6 BITS CHARACTER'
```

280722-39

```
write '                  - 1 1  - Rx 8 BITS CHARACTER'
write '                 D5  -   AUTO ENABLES'
write '                          D4  -   ENTER HUNT MODE'
write '                                  D3  -   Rx CRC ENABLE'
write '                                          D2  -   ADDRESS SEARCH MODE(SDLC)'
write '                                                  D1  -   SYNC CHAR. LOAD INHIBIT'
write '                                                          D0  -   Rx ENABLE'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp - gethex
write 'H'
port(wreg) - 3t
port(wreg) - temp
write concat(1Bh,'L',83h),'            PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy-ci
end



                                        *WR4*
                                        *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr4 - DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                            WRITE REGISTER 4'
write concat(1bh,'L',81h),'                            ****************'
write concat(1bh,'L',80h),'___________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'---------------------------------------------------'
write '    D7     D6  -  0 0  -  X1 CLOCK MODE'
write '               -  0 1  -  X16 CLOCK MODE'
write '               -  1 0  -  X32 CLOCK MODE'
write '               -  1 1  -  X64 CLOCK MODE'
write '                  D5     D4  -  0 0  -  8 BIT SYNC CHARACTER'
write '                              -  0 1  -  16 BIT SYNC CHARACTER'
write '                              -  1 0  -  SDLC MODE(01111110 FLAG)'
write '                              -  1 1  -  EXTERNAL SYNC MODE '
write '                               D3     D2  -  0 0  -  SYNC MODES ENABLE'
write '                                           -  0 1  -  1 STOP BIT/CHARACTER'
write '                                           -  1 0  -  1.5 STOP BITS/CHAR.'
write '                                           -  1 1  -  2 STOP BITS/CHAR.'
write '                                          D1  -   PARITY EVEN/ODD'
write '                                                  D0  -   PARITY ENABLE'
write 'NOTE: Interrupt Vector is modified in B Channel'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp - gethex
write 'H'
port(wreg) - 4t
port(wreg) - temp
write concat(1Bh,'L',83h),'            PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy-ci
end



                                        *WR5*
                                        *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr5- DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                            WRITE REGISTER 5'
write concat(1bh,'L',81h),'                            ****************'
write concat(1bh,'L',80h),'___________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'---------------------------------------------------'
write '    D7  -   DTR'
write '           D6  -   TWO CLOCKS MISSING'
write '                  D5     D4  -  0 0  -  Tx 5 BITS(OR LESS)/CHARACTER'
write '                              -  0 1  -  Tx 7 BITS/CHARACTER'
```

280722-40

```
write '                                  = 1 0  =  Tx 6 BITS/CHARACTER'
write '                                  = 1 1  =  Tx 8 BITS/CHARACTER'
write '                                  D3  =   Tx ENABLE'
write '                                      D2  =   (NOT)SDLC/CRC-16'
write '                                          D1  =   RTS'
write '                                              D0  =   Tx CRC ENABLE'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 5t
port (wreg) = temp
write concat(1BH,'L',83H),'           PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



                                 *WR6*
                                 *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr6 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                         WRITE REGISTER 6'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),'______________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'----------------------------------------------'
write '   SYNC7 SYNC6 SYNC5 SYNC4 SYNC3 SYNC2 SYNC1 SYNC0     MONOSYNC 8 BITS'
write '   SYNC1 SYNC0 SYNC5 SYNC4 SYNC3 SYNC2 SYNC1 SYNC0     MONOSYNC 8 BITS'
write ' '
write '   SYNC7 SYNC6 SYNC5 SYNC4 SYNC3 SYNC2 SYNC1 SYNC0     BISYNC 16 BITS'
write '   SYNC3 SYNC2 SYNC1 SYNC0   1     1     1     1       BISYNC 12 BITS'
write ' '
write '   ADR7  ADR6  ADR5  ADR4  ADR3  ADR2  ADR1  ADR0      SDLC'
write '   ADR7  ADR6  ADR5  ADR4   1     1     1     1        SDLC(ADDRESS 0)'
write ''
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'h'
port(wreg) = 6t
port(wreg) = temp
write concat(1Bh,'L',83h),'           PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



                                 *WR7*
                                 *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr7 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                         WRITE REGISTER 7'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),'______________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'----------------------------------------------'
write '  SYNC7   SYNC6  SYNC5  SYNC4  SYNC3  SYNC2  SYNC1 SYNC0      MONOSYNC 8 BITS'
write '  SYNC5   SYNC4  SYNC3  SYNC2  SYNC1  SYNC0    1     1        MONOSYNC 8 BITS'
write ' '
write '  SYNC5   SYNC14 SYNC13 SYNC12 SYNC11 SYNC10 SYNC9 SYNC8      BISYNC 16 BITS'
write '  SYNC11  SYNC10 SYNC9  SYNC8  SYNC7  SYNC6  SYNC5 SYNC4      BISYNC 12 BITS'
write ' '
write '    0       1      1      1      1      1      1     0      SDLC'
write ''
write concat(1bh,'L',83h),'   PLEASE YTPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
```

280722–41

```
write 'n'
port(wreg) = 7t
port(wreg) = temp
write concat(1Bn,'L',83n),'          PLEASE nIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
yyy=ci
end



                                   *WR8*
                                   *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr8 = DO
define CnAR yyy
define BYTE temp
curhome;cleareos
write concat(1bh,'L',81h),'                         WRITE REGISTER 8'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'| D7  | D6  | D5  | D4  | D3  | D2  | D1  | D0  |'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write ' '
write ' '
write ' '
write ' '
write ' '
write ' '
write ' '
write ' '
write concat(1bh,'L',83h),'   PLEASE TYPE TnE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 2t
port(wreg) = temp
write concat(1Bn,'L',83n),'          PLEASE nIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
yyy=ci
end



                                   *WR9*
                                   *****

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr9= DO
define CnAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                         WRITE REGISTER 9'
write concat(1bh,'L',81h),'                         ****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'| D7  | D6  | D5  | D4  | D3  | D2  | D1  | D0  |'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write '    D7     D6  =  0 0  =  NO RESET'
write '               =  0 1  =  CHANNEL RESET B'
write '               =  1 0  =  CnANNEL RESET A'
write '               =  1 1  =  FORCE HARDWARE RESET'
write '                D5  =  0'
write '                      D4  =   STATUS HIGH/(NOT)STATUS LOW'
write '                            D3  =   MIE'
write '                                  D2  =   DLC'
write '                                        D1  =   NV'
write '                                              D0  =   VIS'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'n'
port(wreg) = 9t
port(wreg) = temp
write concat(1Bn,'L',83n),'          PLEASE nIT ANY KEY TO RETURN'
write concat(1bh,'1',80h), ' '
yyy=ci
end
```

280722-42

```



                                       *WR10*
                                       ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr10 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                     WRITE REGISTER 10'
write concat(1bh,'L',81h),'                     *****************'
write concat(1bh,'L',80h),'_________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------'
write '    D7  =   CRC PRESET I/(NOT)O'
write '          D6     D5  =  0 0  =  NRZ'
write '                     =  0 1  =  NRZI'
write '                     =  1 0  =  FM1 (TRANSMISSION 1)'
write '                     =  1 1  =  FM0 (TRANSMISSION 0)'
write '                      D4  =  GO ACTIVE ON ROLL'
write '                            D3  =  MARK/(NOT)FLAG IDEL'
write '                                  D2  =  ABORT/(NOT)FLAG ON UNDERRRUN'
write '                                        D1  =  LOOP MODE  __________'
write '                                              D0  =  6 BIT/8 BIT SYNC'
write concat(1bh,'L',83h),'  PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 10t
port(wreg) = temp
write concat(1Bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end




                                       *WR11*
                                       ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr11 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                     WRITE REGISTER 11'
write concat(1bh,'L',81h),'                     *****************'
write concat(1bh,'L',80h),'_________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------'
write '    D7  =   (NOT)RTxC XTAL/(NOT)NO XTAL        ____'
write '          D6     D5  =  0 0   =  RECEIVE CLOCK = RTxC PIN'
write '                     =  0 1   =  RECEIVE CLOCK = (NOT)TRxC PIN'
write '                     =  1 0   =  RECEIVE CLOCK = BR GENERATOR OUTPUT'
write '                     =  1 1   =  RECEIVE CLOCK = DPLL OUTPUT'
write '                      D4     D3  =  0 0  =  TRANSMIT CLOCK = (NOT)RTxC PIN'
write '                                 =  0 1  =  TRANSMIT CLOCK = (NOT)TRxC PIN'
write '                                 =  1 0  =  TRANSMIT CLOCK = BR GEN. OUTPUT'
write '                                 =  1 1  =  TRANSMIT CLOCK = DPLL OUTPUT'
write '                                  D2  =  (NOT)TRxC 0/1'
write '                                        D1      D0 = 0 0 = XTAL OUTPTU'
write '                                                   = 0 1 = TRANSMIT CLOCK'
write '                                                   = 1 0 = BR GENERATOR O/P'
write '                                                   = 1 1 = DPLL OUTPUT'
write concat(1bh,'L',83h),'   PLEASE YTPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h), ' '
temp = gethex
write 'H'
port(wreg) = 11t
port(wreg) = temp
write concat(1Bh,'L',83h),'          PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



```

280722-43

```
                                        *WR12*
                                        ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr12= DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                          WRITE REGISTER 12'
write concat(1bh,'L',81h),'                          *****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write '     D7  =   TC7'
write '             D6  =   TC6'
write '                     D5  =   TC5'
write '                             D4  =   TC4'
write '                                     D3  =   TC3'
write '                                             D2  =   TC2'
write '                                                     D1  =   TC1'
write '                                                             D0  =   TC0'
write ' '
write 'NOTE: Lower Byte of TIME CONSTANT'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h), ' '
temp = gethex
write 'n'
port(wreg) = 12t
port(wreg) = temp
write concat(1bh,'L',83H),'           PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



                                        *WR13*
                                        ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr13= DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                          WRITE REGISTER 13'
write concat(1bh,'L',81h),'                          *****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write '     D7  =   TC15'
write '             D6  =   TC14'
write '                     D5  =   TC13'
write '                             D4  =   TC12'
write '                                     D3  =   TC11'
write '                                             D2  =   TC10'
write '                                                     D1  =   TC9'
write '                                                             D0  =   TC8'
write ' '
write 'NOTE: Upper Byte of TIME CONSTANT'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 13t
port(wreg) = temp
write concat(1BH,'L',83H),'           PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end



                                        *WR14*
                                        ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
```

280722-44

```
define PROC wr14 = DO
define CHAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                           WRITE REGISTER 14'
write concat(1bh,'L',81h),'                           *****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write '     D7     D6     D5  =  0 0 0  =  NULL COMMAND'
write '                       =  0 0 1  =  ENTER SEARCn MODE'
write '                       =  0 1 0  =  RESET MISSSING CLOCK'
write '                       =  0 1 1  =  DISABLE DPLL'
write '                       =  1 0 0  =  SET SOURCE = BR GENERATOR'
write '                       =  1 0 1  =  SET SOURCE = (NOT)RT x C'
write '                       =  1 1 0  =  SET FM MODE'
write '                       =  1 1 1  =  SET NRZI MODE'
write '                       D4  =   LOCAL LOOPBACK'
write '                             D3  =   AUOT ECHO'
write '                                   D2  =   (NOT)DTR REQUEST FUNCTION'
write '                                         D1  =   BR GENERATOR SOURCE'
write '                                               D0  =   BR GEN. ENABLE'
write concat(1bh,'L',83h),'  PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'n'
port(wreg) = 14t
port(wreg) = temp
write concat(1Bn,'L',83n),'           PLEASE nIT ANY KEY TO RETURN '
write concat(1bh,'1',80h), ' '
yyy=ci
end
curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr14 = DO
define CnAR yyy
define BYTE temp
curhome ;cleareos
write concat(1bh,'L',81h),'                           WRITE REGISTER 14'
write concat(1bh,'L',81h),'                           *****************'
write concat(1bh,'L',80h),'_____________________________________________________________'
write concat(1bh,'L',80h),'¶ D7  ¶ D6  ¶ D5  ¶ D4  ¶ D3  ¶ D2  ¶ D1  ¶ D0  ¶'
write concat(1bh,'L',80h),'-------------------------------------------------------------'
write '     D7     D6     D5  =  0 0 0  =  NULL COMMAND'
write '                       =  0 0 1  =  ENTER SEARCH MODE'
write '                       =  0 1 0  =  RESET MISSSING CLOCK'
write '                       =  0 1 1  =  DISABLE DPLL'
write '                       =  1 0 0  =  SET SOURCE = BR GENERATOR'
write '                       =  1 0 1  =  SET SOURCE = (NOT)RT x C'
write '                       =  1 1 0  =  SET FM MODE'
write '                       =  1 1 1  =  SET NRZI MODE'
write '                       D4  =   LOCAL LOOPBACK'
write '                             D3  =   AUOT ECHO'
write '                                   D2  =   (NOT)DTR REQUEST FUNCTION'
write '                                         D1  =   BR GENERATOR SOURCE'
write '                                               D0  =   BR GEN. ENABLE'
write concat(1bh,'L',83h),'  PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'n'
port(wreg) = 14t
port(wreg) = temp
write concat(1Bn,'L',83n),'           PLEASE nIT ANY KEY TO RETURN '
write concat(1bh,'1',80h), ' '
yyy=ci
end



                               *WR15*
                               ******

curhome;cleareos
write concat(1bh,'L',83h),'LOADING....'
write concat(1bh,'L',80h),''
define PROC wr15= DO
define CHAR yyy
define BYTE temp
curhome;cleareos
```

280722–45

```
write concat(1bh,'L',81h),'                                WRITE REGISTER 15'
write concat(1bh,'L',81h),'                                *****************'
write concat(1bh,'L',80h),'_______________________________________________________________
write concat(1bh,'L',80h),'| D7  | D6  | D5  | D4  | D3  | D2  | D1  | D0  |
write concat(1bh,'L',80h),'----------------------------------------------------------------
write '     D7  =   BREAK/ABORT IE'
write '              D6  =   Tx UNDERRUN/EOM IE'
write '                        D5  =   CTS IE'
write '                                 D4  =   SYNC/HUNT IE'
write '                                           D3  =   CD IE'
write '                                                     D2  =   0'
write '                                                               D1  =   ZERO COUNT IE'
write '                                                                         D0  =   0'
write concat(1bh,'L',83h),'   PLEASE TYPE THE VALUE TO BE WRITTEN and <cr>'
write concat(1bh,'L',80h),' '
temp = gethex
write 'H'
port(wreg) = 15t
port(wreg) = temp
write concat(1BH,'L',83H),'           PLEASE HIT ANY KEY TO RETURN'
write concat(1bh,'l',80h), ' '
yyy=ci
end
```

280722-46

APPLICATION NOTE

AP-273

November 1986

Developing MCS®-96 Applications Using the SBE-96

DAVE SCHOEBEL
DSO APPLICATIONS

Order Number: 280249-001

INTRODUCTION

With the increasing demands of industrial and computer control applications, today's designers are looking for solutions whose performance extends beyond that of conventional 8-bit architectures. Traditionally, these control system architects must depend upon expensive and complex multi-chip microprocessors to achieve this high performance, but now a 16-bit single chip microcontroller can offer a much more cost-effective solution. Microcontrollers are microprocessors specially configured to monitor and control mechanisms and processes rather than manipulate data. They include CPU, program memory, data memory and a array of specialized peripherals on chip to produce a low component count solution. The MCS-96 family uses 120,000 transistors to implement a high performance 16-bit CPU, 8K bytes of program memory, 232 bytes of data memory and both analog and digital I/O features. Supporting this device are a suite of development tools hosted on both Intel development systems (Series III and IV) and industry standard hosts (IBM PC XT and PC AT).

This application note includes a brief description of the MCS-96 family of microcontrollers, its software development environment and hardware debugging centered around the iSBE-96 Single Board Emulator. Also included are helpful hints and programs to enable you to get the most from your investment dollars. The application note is partitioned into two sections. The first secion introduces the MCS-96 architecture and development environment while the later section provides indepth details of the iSBE-96 including its customization to your particular environment.

MCS®-96 MICROCONTROLLER OVERVIEW

Introduction to the MCS®-96 Architecture

The MCS-96 architecture consists of a 16-bit central processing unit (CPU) and a multitude of peripheral and I/O functions integrated into a single silicon component as shown in Figure 1. The CPU supports bit, byte and word operations. Double words (32-bits) are also supported in a subset of the instruction set. With a 12 MHz input crystal frequency, the MCS-96 microcontroller can perform a 16-bit addition in 1.0 microseconds (μs) and a 16 x 16 bit multiply or 32/16 bit divide in just 6.5 μs.

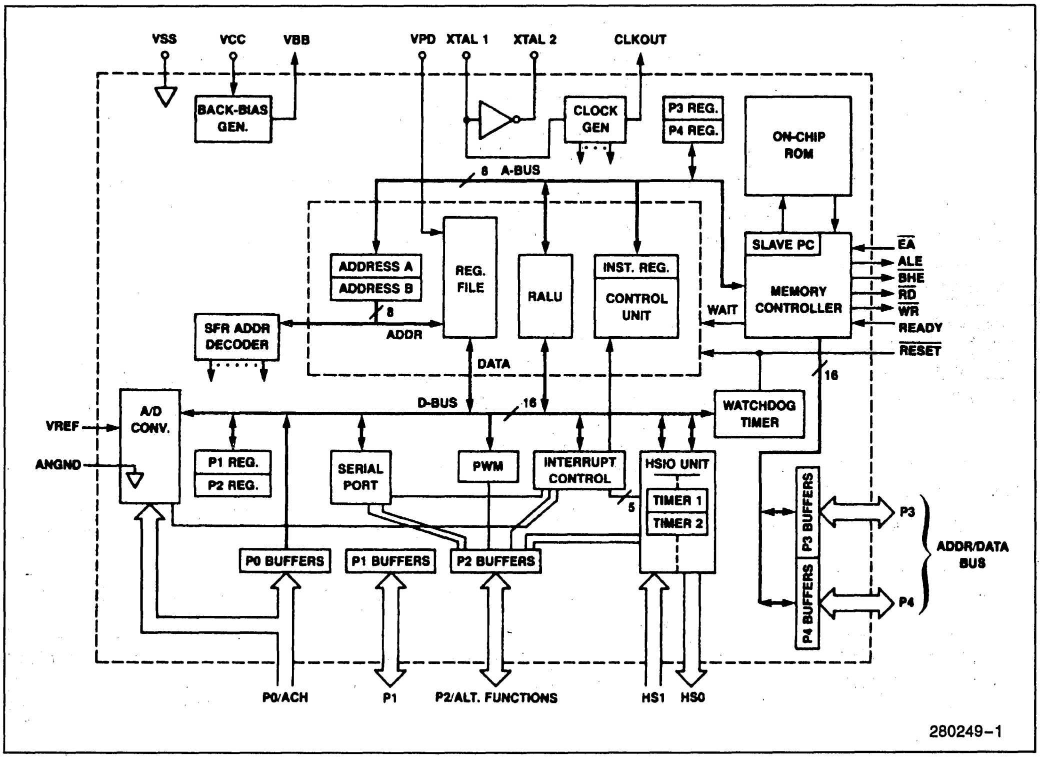


Figure 1. Block Diagram of the MCS®-96 Microcontroller

There are four high-speed trigger inputs that can record the times at which external events occur as often as every 2 μs (at 12 MHz crystal frequency). Up to six high-speed pulse generator outputs can trigger external events at pre-selected times. Additionally, the high-speed output unit can simultaneously perform timer functions. Up to four 16-bit software timers can be in operation simultaneously, in addition to the two 16-bit hardware timers. This makes the MCS-96 microcontroller particularly useful in process and control applications.

There is an optional on-chip analog to digital (A/D) converter which can convert up to four (in the 48-pin version) or eight (in the 68-pin version) analog input channels (10-bits resolution) in only 22 μs for the 8x9xBH parts or 42 μs for the 8x9x-90 parts.

Also provided on-chip is a full duplex serial port, dedicated baud rate generator, 16-bit watchdog timer, and a pulsewidth modulated output signal. Table 1 shows the features summary for the MCS-96 microcontroller. Table 2 shows the different configurations for the MCS-96 family of microcontrollers.

The following sections briefly describe some of the features of the MCS-96 microcontroller.

High Speed I/O Unit (HSIO)

The HSIO unit consists of the High-Speed Input unit (HSI), the High-Speed Output unit (HSO), one counter, and one timer. The "high-speed" means that the units can perform functions based on the timers without CPU intervention. The HSI unit records times when events occur and the HSO unit triggers events at preprogrammed times. All actions within the HSIO units are synchronized to the timer or counter.

The HSI unit can detect transitions on any of its four input lines. When one occurs, it records the time from Timer 1 and which input lines made the transition. The time is recorded with a 2 μs resolution and is stored in an eight-level first-in-first-out buffer (FIFO). The unit can activate an interrupt when the holding register is loaded or the 6th entry to the FIFO has been made.

Table 1. MCS®-96 Microcontroller Features and Benefits Summary

Features	Benefits
16-Bit CPU	Efficient machine with higher throughput.
8K Bytes ROM	Large program space for more complex, larger programs.
232 Bytes RAM	Large on-board register file.
Hardware MUL/DIV	Provides good math capability 16 by 16 multiply or 32 by 16 divide in 6.5 μs @ 12 MHz.
6 Addressing Modes	Provides greater flexibility of programming and data manipulation.
High Speed I/O Unit 4 dedicated I/O lines 4 programmable I/O lines	Can measure and generate pulses with high resolution (2 μs @ 12 MHz).
10-Bit A/D Converter	Reads the external analog inputs.
Full Duplex Serial Port	Provides asynchronous serial link to other processors or systems.
Up to 40 I/O Pins	Provides TTL compatible digital data I/O including system expansion with standard 8 or 16-bit peripherals.
Programmable 8 Source Priority Interrupt System	Respond to asynchronous events
Pulse Width Modulated Output	Provides a programmable pulse train with variable duty cycle. Also used to generate analog output.
Watchdog Timer	Provides ability to recover from software malfunction or hardware upset.
48 Pin (DIP) & 68-Pin (Flatpack, Pin Grid Array) Versions	Offers a variety of package types to choose from to better fit a specific application need for number of I/Os and package size.

The HSO unit can be programmed to set or clear any of its six output lines, reset timer 2, trigger an A/D conversion, or set one of four software timer flags at a selected time. An interrupt can be enabled for any of these events and either Timer 1 or Timer 2 can be referenced for the programmed time value. Also, up to eight commands for preset actions can be stored in the Content Addressable Memory (CAM) file. After each action is carried out at the preset time, the command is removed from the CAM, making room for another command. The CPU is kept informed with a status bit that indicates if there is room for another command in the CAM.

A/D Converter

The analog-to-digital (A/D) converter is a 10-bit, successive approximation converter with an internal sample and hold circuit. It has a fixed conversion time of 88 CPU state times. A state time is one complete crystal frequency period. With a 12 MHz crystal, a state time is 250 nanoseconds (ns) so the conversion will take 22 μs.

The analog input needs to be in the range of 0 to VREF (nominally VREF = 5V) and can be selected from any of the eight analog input lines. The conversion is then initiated by either setting the control bit in the A/D command register or by programming the HSO unit to trigger the conversion at some specified time.

Serial Port

The on-chip serial port is compatible with the MCS-51 family (8051, 8031, etc.) serial port. It is a full duplex port and there is double-buffering on receive. Additionally, the serial port supports three asynchronous modes and one synchronous mode of operation. With the asynchronous modes eight or nine bits of data can be selected and even parity can optionally be inserted for one of the data bits. Selectable interrupts for transmit ready, receive ready, ninth data bit received, and parity error provide support for a variety of interprocessor communications protocols.

Baud rates in all modes are determined by an independent 16-bit on-chip baud rate generator. The input to the baud rate generator can come from either the XTAL1 or the T2CLK pins. The maximum baud rate provided by the generator in asynchronous mode is 187.5K baud and in synchronous mode is 1.5M baud.

Watchdog Timer

The watchdog timer is a 16-bit counter which, once started, is incremented every state time. The watchdog timer is optionally started, and once started it cannot be stopped unless the system is reset. To start or clear the watchdog timer simply write a 1EH followed by a 0E1H to the WDT register (address 0AH). If not cleared before it overflows, the watchdog timer will pull the RESET pin low for two state times, causing the system to be reinitialized. With a 12 MHz crystal, the watchdog timer will overflow after 16 milliseconds (ms).

The watchdog timer is provided as a means of graceful recovery from a software upset. The counter must be cleared by the software before it overflows or the timer assumes that an upset has occurred and activates the RESET pin. Since the watchdog timer cannot be turned off by software, the system is protected against the upset inadvertently disabling the watchdog timer. The watchdog timer has also been designed to maintain its state through power glitches on VCC. The glitches can be as low as 0V or as high as 7V for as long as 1 μs to 1 ms.

Pulse Width Modulator (PWM)

The PWM output can produce a pulse train having a fixed period of 256 state times and a programmable width of zero to 255 state times. The width is programmed by loading the desired value, in state times, to the PWM control register.

Table 2. Configurations of the MCS®-96 Family of Microcontrollers

Options		68 Pin	48 Pin
Digital I/O	ROMLESS	8096	8094
	EPROM	8796	8794
	ROM	8396	8394
Analog and Digital I/O	ROMLESS	8097	8095
	EPROM	8797	8795
	ROM	8397	8395

Memory Space

The addressable memory space of the MCS-96 microcontroller consists of 64K bytes. Although most of this space is available for general use, some locations have special purposes (0000H through 00FFH and 1FFEH through 207FH). All other locations can be used for either program or data storage or for memory mapped peripherals. A memory map is shown in Figure 2.

The internal register locations (0000H through 00FFH) on the 8096 are divided into two groups, a register file and a set of Special Function Registers (SFRs).

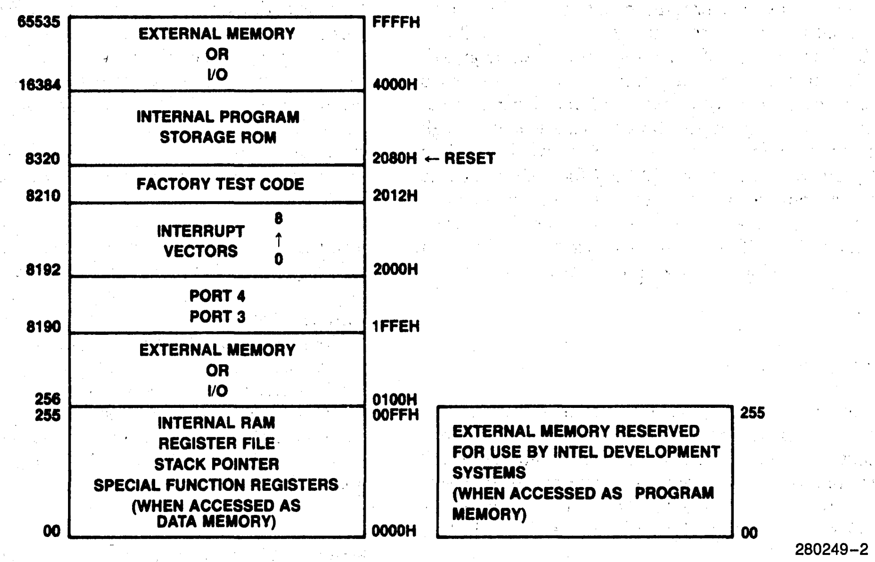


Figure 2. Memory Map

REGISTER FILE

Locations 1AH through 0FFH contain the register file. The register file memory map is shown in Figure 3. Additionally, locations 0F0H through 0FFH can be powered separately so that they will retain their contents when power is removed from the 8096 VCC pin. There are no restrictions on the use of the register file except that code cannot be executed from it. If an attempt to execute instructions from locations 00H through 0FFH is made, the instructions will be fetched from external memory. This section of external memory is reserved for use by Intel development tools. Execution of a nonmaskable interrupt (NMI) will force a call to external location 0000H, therefore, the NMI is also reserved for Intel development tools.

SPECIAL FUNCTION REGISTERS (SFRs)

Locations 00H through 17H are used to access the SFRs. Locations 18H and 19H contain the stack pointer. All of the I/O on the 8096 is controlled through the SFRs. Many of these registers serve two functions; one if they are read from, the other if they are written to. Figure 3 shows the locations and names of these registers. A summary of the capabilities of each of these registers is shown in Figure 4. Note that these registers can be accessed only as bytes unless otherwise indicated. The stack pointer must be initialized by the user program and can point anywhere in the 64K memory space. The stack builds down, that is, it is a post-increment (POP), pre-decrement (PUSH) stack.

RESERVED MEMORY SPACES

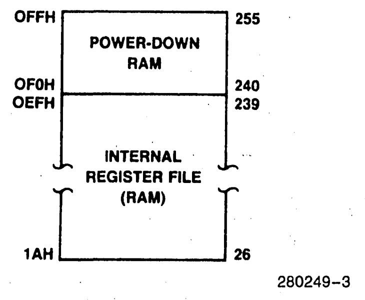
Locations 1FFEH and 1FFFH are reserved for Ports 3 and 4 respectively. This enables easy reconstruction of these ports if external memory is used in the system. This also simplifies changing between the ROMless, EPROMed, and ROMed parts without changing the program addresses for ports 3 and 4. If ports 3 and 4 are not going to be reconstructed, these locations can be treated as any other external memory location.

The nine interrupt vectors are stored in locations 2000H through 2011H. The ninth vector (2010H–2011H) is reserved for Intel development systems. Figure 5 shows the interrupt vector locations and priority. When enabled, an interrupt occurring on any of these sources will force a call to the location stored in the vector location for that interrupt source. Internal locations 2012H through 207FH are reserved for Intel's factory test code and for use by future components. To ensure compatibility with future parts, external locations 2012H through 207FH (if present) should contain the hex value FFH.

SOFTWARE DEVELOPMENT OVERVIEW

MCS®-96 Microcontroller Software Development Packages

The MCS-96 Microcontroller Software Support Package provides 8096 development system support specifically designed for the MCS-96 family of single chip microcontrollers. The package consists of a symbolic macro assembler (ASM-96), Linker/Relocator (RL-96), Floating Point Arithmetic Library (FPAL96) and the librarian (LIB-96).



19H 18H	STACK POINTER	STACK POINTER	25 24
17H		PWM_CONTROL	23
16H	IOS1	IOC1	22
15H	IOS0	IOC0	21
14H 13H 12H	RESERVED	RESERVED	20 19 18
11H	SP_STAT	SP_CON	17
10H	IO PORT 2	IO PORT 2	16
0FH	IO PORT 1	IO PORT 1	15
0EH	IO PORT 0	BAUD_RATE	14
0DH	TIMER2 (HI)	RESERVED	13
0CH	TIMER 2 (LO)		12
0BH	TIMER 1 (HI)		11
0AH	TIMER 1 (LO)	WATCHDOG	10
09H	INT_PENDING	INT_PENDING	9
08H	INT_MASK	INT_MASK	8
07H	SBUF (RX)	SBUF (TX)	7
06H	HSI_STATUS	HSO_COMMAND	6
05H	HSI_TIME (HI)	HSO_TIME (HI)	5
04H	HSI_TIME (LO)	HSO_TIME (LO)	4
03H	AD_RESULT (HI)	HSI_MODE	3
02H	AD_RESULT (LO)	AD_COMMAND	2
01H	R0 (HI)	R0 (HI)	1
00H	R0 (LO)	R0 (LO)	0
	(WHEN READ)	(WHEN WRITTEN)	

Figure 3. Register File Memory Map

The PL/M-96 Software Package provides 8096 high-level language development system suport. The package consists of a structured high-level language compiler (PL/M-96), Linker/Relocator (RL-96), Floating Point Arithmetic Library (FPAL96) and the librarian (LIB-96).

Both software packages run on the IBM PC XT and AT (with DOS 3.0 or greater) and on Series III/IV Intellec® development systems.

A detailed description of the tools contained in the packages is given in the following sections.

ASM-96 MACRO ASSEMBLER

The 8096 macro assembler translates the symbolic assembly language instructions into relocatable object code. Since the object modules are linkable and locatable, ASM-96 encourages modular programming practices. The macro facility in ASM-96 enables programmers to save development and maintenance time, since common code sequences only have to be done once. The assembler also provides conditional assembly capabilities. ASM-96 supports symbolic access to the many features of the 8096 architecture as described previously. A file is provided with all of the 8096 hardware registers defined. Alternatively, the user can define any subset of the 8096 hardware register set. Math routines are supported with instructions for 16 x 16-bit multiply or 32/16-bit divide.

Modular programs divide a rather complex program into smaller functional units that are easier to code, to debug, and to change. The separate modules can then be linked and located as desired into one program module of executable code. Standard modules can be developed and used in different applications thus saving software development time.

PL/M-96

PL/M-96 is a structured, high-level programming language used for developing software for the Intel MCS-96 family of microcontrollers. Symbolic access to the on-chip resources of the MCS-96 microcontroller is provided in PL/M-96. The PL/M-96 compiler translates the PL/M-96 language into 8096 relocatable object code, compatible with object code generated by other MCS-96 translators (such as ASM-96). This enables improved programmer productivity and application reliability. PL/M-96 has been efficiently designed to map into the machine architecture, so as not to trade off higher programmer productivity with inefficient code. PL/M-96 is also compatible with PL/M-86 thus assuring design portability and minimal learning effort for programmers already familiar with PL/M.

COMBINING PL/M-96 AND ASM-96

For each procedure activation (CALL statement or function reference) in the source, the object code uses a calling sequence. The calling sequence places the procedure's actual parameters (if any) on the stack, then activates the procedure with a CALL instruction. The parameters are placed on the stack in left to right order. Since the direction of stack growth is from higher locations to lower, the first parameter occupies the highest position on the stack and the last parameter occupies

Register	Description
R0	Zero Register—Always read as a zero, useful for a base when indexing and as a constant for calculations and compares.
AD_RESULT	A/D Result Hi/Low—Low and high order Results of the A/D converter (byte read only)
AD_COMMAND	A/D Command Register—Controls the A/D
HSI_MODE	HSI Mode Register—Sets the mode of the High Speed Input unit.
HSI_TIME	HSI Time Hi/Lo—Contains the time at which the High Speed Input unit was triggered. (word read only)
HSO_TIME	HSO Time Hi/Lo—Sets the time for the High Speed Output to execute the command in the Command Register. (word write only)
HSO_COMMAND	HSO Command Register—Determines what will happen at the time loaded into the HSO Time registers.
HSI_STATUS	HSI Status Registers—Indicates which HSI pins were detected at the time in the HSI Time registers.
SBUF (TX)	Transmit buffer for the serial port, holds contents to be output.
SBUF (RX)	Receive buffer for the serial port, holds the byte just received by the serial port.
INT_MASK	Interrupt Mask Register—Enables or disables the individual interrupts.
INT_PENDING	Interrupt Pending Register—Indicates when an interrupt signal has occurred on one of the sources.
WATCHDOG	Watchdog Timer Register—Written to periodically to hold off automatic reset every 64K state times.
TIMER1	Timer 1 Hi/Lo—Timer 1 high and low bytes. (word read only)
TIMER2	Timer 2 Hi/Lo—Timer 2 high and low bytes. (word read only)
IOPORT0	Port 0 Register—Levels on pins of port 0.
BAUD_RATE	Register which contains the baud rate, this register is loaded sequentially.
IOPORT1	Port 1 Register—Used to read or write to Port 1.
IOPORT2	Port 2 Register—Used to read or write to Port 2.
SP_STAT	Serial Port Status—Indicates the status of the serial port.
SP_CON	Serial port control—Used to set the mode of the serial port.
IOS0	I/O Status Register 0—Contains information on the HSO status.
IOS1	I/O Status Register 1—Contains information on the status of the timers and of the HSI.
IOC0	I/O Control Register 0—Controls alternate functions of HSI pins, Timer 2 reset sources and Timer 2 clock sources.
IOC1	I/O Control Register 1—Controls alternate functions of Port 2 pins, timer interrupts and HSI interrupts.
PWM_CONTROL	Pulse Width Modulation Control Register—Sets the duration of the PWM pulse.

Figure 4. SFR Summary

the lowest position. Note that a BYTE or SHORTINT parameter value occupies two bytes on the stack, with the value in the lower (even address) byte. The contents of the higher byte are undefined. A parameter of type WORD or INTEGER (16 bits) is pushed as a word. A parameter of type DWORD, LONGINT or REAL (32 bits) is pushed as two words; the high-order word is pushed first.

Source	Vector Location (High Byte)	Vector Location (Low Byte)	Priority
Software	2011H	2010H	Not Applicable
Extint	200FH	200EH	7 (Highest)
Serial Port	200DH	200CH	6
Software Timers	200BH	200AH	5
HSI.0	2009H	2008H	4
High Speed Outputs	2007H	2006H	3
HSI Data Available	2005H	2004H	2
A/D Conversion Complete	2003H	2002H	1
Timer Overflow	2001H	2000H	0 (Lowest)

Figure 5. Interrupt Vector Locations

After the parameters are passed, the CALL instruction places the return address on the stack. Function results are returned via a global PL/M-96 double-word register, PLM$REG located at 1CH. If a byte value is returned, the low-order byte is used; if a word value is returned, the low-order word is used; otherwise, the full register is used. PL/M-96 uses the eight byte registers at addresses 1CH–23H for temporary computations. The library PLM-96LIB defines the public symbol PLM$REG.

Table 3 describes symbol type matching between a PL/M-96 global variable and an ASM-96 global variable. Note that except for NULL, no matches occur between any ASM-96 type stamp and the PL/M-96 type stamps ARRAY and STRUCTURE. A mismatch warning can be prevented by attaching the type stamp NULL to the variable in question in the ASM-96 module.

The easiest way to ensure compatibility between PL/M-96 programs or procedures and ASM-96 subroutines is simply to write a dummy procedure in PL/M-96 with the same argument list as the desired assembly language subroutine and with the same attributes. Then, compile the dummy procedure with the specified CODE control. This will produce a pseudo-assembly listing of the generated MCS-96 code, which can then be copied as the prologue and epilogue of the assembly language subroutine.

OTHER SOFTWARE DEVELOPMENT TOOLS

The RL96 linker and relocator program is a utility that performs two functions useful in MCS-96 software development. First, the link function combines a number of object modules generated by ASM-96, PL/M-96, and system libraries (such as PLM96.lib and FPAL96.lib) into a single program. Secondly, the locate function assigns an absolute address to all relocatable addresses in the linked MCS-96 object module. RL96 resolves all external symbol references between modules and will select object modules from library files if necessary. besides the absolute object module file, RL96 produces a listing file that shows the results of the link/locate, including a memory map symbol table and an optional cross reference listing. With the relocator the programmer can concentrate on software functionality and not worry about the absolute addresses of the object code. All program symbols are passed through into the object module as debug records. The FPAL96 floating point arithmetic library contains single precision 32-bit floating point arithmetic functions. All math complies with the IEEE floating point standard for accuracy and reliability. FPAL96 includes the basic arithmetic operations (i.e., add, subtract, multiply, divide, mod, square root) and other widely used operations (i.e., compare, negate, absolute, remainder). An error handler is included to handle exceptions commonly encountered during arithmetic operations such as divide by zero.

The LIB96 utility creates and maintains libraries of software object modules. The user can develop standard modules and place them in libraries. Application programs can then call these modules using predefined interfaces. LIB96 has a streamlined set of commands (create, add, delete, list, exit) to provide ease of use. When using object libraries, RL906 will only include those object modules that are required to satisfy external references, thus saving memory space.

Table 3. ASM96-PL/M-96 Symbol Type Matching

PL/M-96 / ASM96	Byte	Word	Dword	Short Int	Integer	Long Int	Real	Array	Structure	Label	Procedure
BYTE	M			M							
WORD		M			M						
LONG			M			M					
REAL							M				
ENTRY										M	M
NULL	M	M	M	M	M	M	M	M	M	M	M

iSBE-96 EMULATOR OVERVIEW

Introduction to the iSBE-96 Emulator

The iSBE-96 Single Board Emulator supports the execution and debugging of programs for the MCS-96 family of microcontrollers at speeds up to 12 MHz. Figure 6 shows a block diagram of the iSBE-96 emulator. The iSBE-96 emulator consists of an 8097 microcontroller, a 12 MHz execution clock, 16K of zero wait state RAM memory, and a user cable which connects the MCS-96 pin functions to the user's prototype system. The iSBE-96 emulator also supports an 8096 extended address/data bus for users with off chip memory and reconstructs port 3 and 4 for the users of the ROMed parts, 839x, and the EPROM parts, 879x. Additionally, the iSBE-96 emulator provides two RS-232 serial ports, serial communications cable, an EPROM based monitor for fundamental emulator control and functionality, and a software program for interfacing to a host computer. Intel currently supports an IBM PC XT and AT, and the Series III/IV Intellec development systems as hosts.

iSBE-96 Emulator I/O

The iSBE-96 emulator's on-board input and output (I/O) devices are used to manage the emulator's resources. These I/O devices are mapped into memory at locations 1F00H through 1FFFH. This memory block (1F00H through 1FFFH) is reserved for use by the iSBE-96 emulator. Table 4 shows the iSBE-96 memory mapped I/O address assignments. Since this memory block is in all possible memory configurations of the iSBE-96 emulator (see Figure 7 for the iSBE-96 memory map), it is possible for user programs to utilize any or all of the system I/O devices.

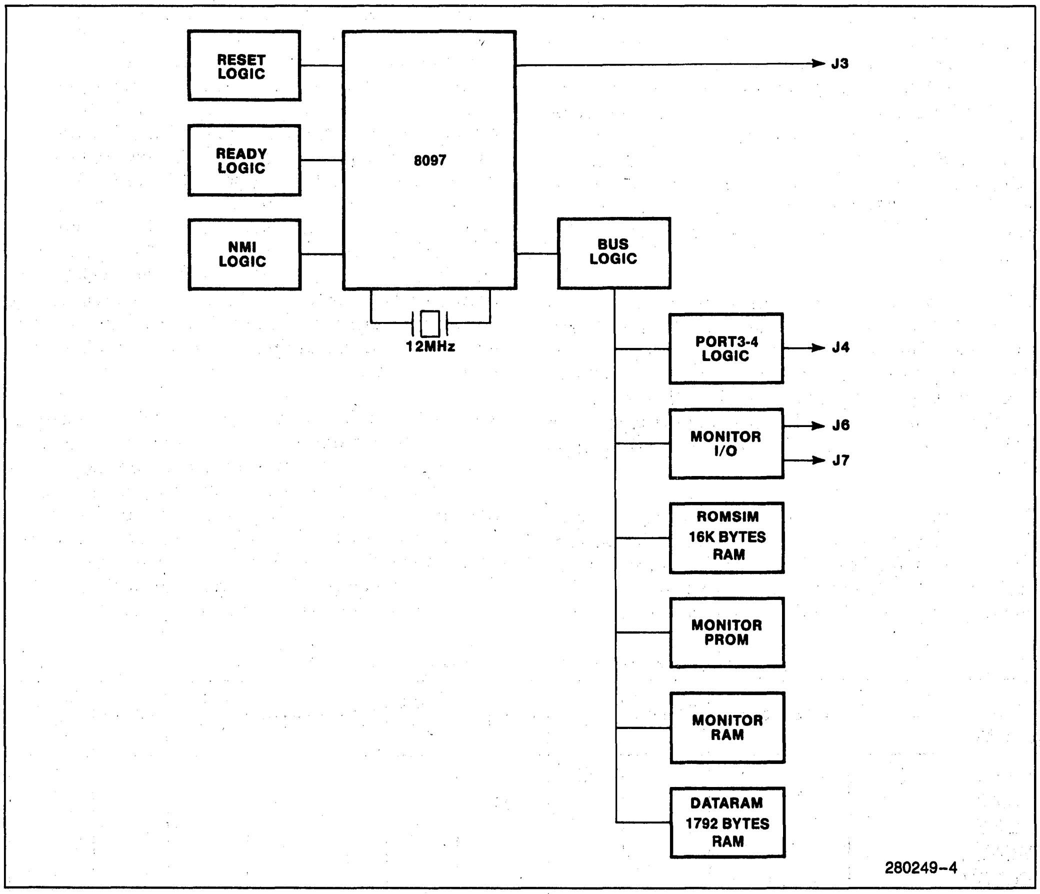


Figure 6. Block Diagram for the iSBE-96 Single Board Emulator

Table 4. iSBE-96 Memory Mapped I/O Address Assignments

Address	Function
01FEO	Data set USART data register
01FE2	Data set USART control/status register
01FE4	Data terminal USART data register
01FE6	Data terminal USART control/status register
01FE8	Timer counter 0
01FEA	Timer counter 1
01FEC	Timer counter 2
01FEE	Timer mode control register
01FF0	iSBE-96 mode register
01FF2	Port 3/4 control register
01FFE	Port 3 reconstruction
01FFF	Port 4 reconstruction

RS-232 SERIAL PORTS

Included as part of the on-board I/O are two RS-232 serial ports. One is configured as Data Communications Equipment (DCE) and the other as Data Terminal Equipment (DTE). When operating with the host software provided with the iSBE-96 emulator, the DCE port is used for the system console and the link for exchanging files. Table 5 shows the pin configuration of the two serial port connectors.

The serial ports are serviced under control of the on-board 8097 non-maskable interrupt (NMI). The NMI has the highest priority of all interrupts on the 8097 microcontroller. While in emulation (user program is executing) the user program will be interrupted if monitor commands are entered from the console. Valid commands input on the console will be executed by the monitor even during emulation. Therefore, the iSBE-96 emulator provides full-speed 12 MHz emulation, only if no commands are entered until emulation is halted.

MCS®-96 PORT 3/4 AND EXTENDED ADDRESS/DATA BUS

With the MCS-96 microcontroller, ports 3 and 4 pins can be used as actual port pins or as an extended address and data bus. For the convenience of the users of the ROMed parts and the EPROMed parts (839x and 879x respectively) the iSBE-96 emulator provides a reconstruction of ports 3 and 4. Additionally, for users of the ROMless parts or parts in external access mode, the iSBE-96 emulator provides an extended address and data bus. The selection of what the port pins are used for is left to the user via the MAP BUSPINS command. On power-up of the iSBE-96 emulator, the default mapping is for port 3/4.

iSBE-96 Emulator Memory Map

The iSBE-96 emulator has a number of memory map options. All of the memory maps are compatible with the MCS-96 microcontroller. Figure 7 shows the different memory map selections available. Each memory map is selected by the MAP MODE command, which changes the memory map currently recognized by the iSBE-96 emulator. Table 6 summarizes the physical memory configurations of the iSBE-96 emulator needed to implement the various memory maps. Note that modes (memory maps) 1 through 3 require that the eight 2K x 8 RAM chips (16K bytes of RAM) on the iSBE-96 emulator be replaced by 8K x 8 RAM or PROM chips.

The memory map is controlled by two bipolar PROMs and an eight bit register (the mode register at 01FF0H). The format of the mode register is shown in Figure 8. The mode register is a write only register and any writes to this register need to be done with caution. In addition to the memory map, the mode register is used to enable each of the five possible sources of interrupts connected to the NMI.

Monitor Command Summary

The iSBE-96 monitor is capable of executing a number of commands without being connected to a host development system. It is possible to connect only a video terminal to the iSBE-96 emulator and still have significant debug capability. Table 7 summarizes the monitor commands. The load and save command will not work with the iSBE-96 emulator connected to a terminal. Load and save requires the iSBE-96 emulator to be connected to a host development system. If a non-Intel suported host is used a software program will need to be written for that computer to provide the mass storage/retrieval access and the proper communications interface protocol to the iSBE-96 emulator.

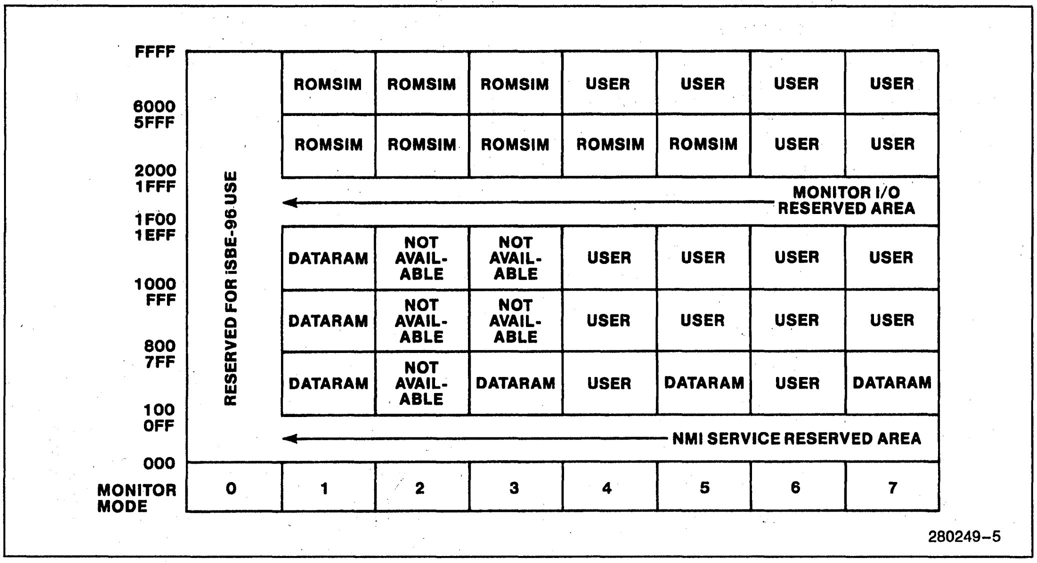


Figure 7. iSBE-96 Memory Map and Monitor Modes

Design Considerations

When debugging MCS-96 designs with the iSBE-96 emulator, there are some features of the emulator that should be considered or taken into account as early in the design process as possible.

MEMORY

The user's prototype memory should be mapped to be compatible with one of the iSBE-96 memory maps (illustrated in Figure 7) or else a new memory map for the iSBE-96 emulator must be generated. External address locations 0000H through 00FFH and locations 1F00H through 1FFFH are reserved for development system use and should not be used when using an Intel emulator.

Program code or memory mapped peripherals should be temporarily relocated before debugging with the iSBE-96 emulator.

Table 5. DS/DT RS-232 Pin-Out Configuration

Pin Number	Signal Name/Connector	
	DCE/J7	DTE/J6
1	GND	GND
2	TXD-I	TXD-O
3	RXD-O	RXD-I
4	RTS-I	RTS-O
5	CTS-O	CTS-I
6	DSR-O	DSR-I
7	GND	GND
20	DTR-I	DTR-O

Table 6. Memory Configurations for Each Mode

Mode	Allowable Memory Configurations
0	Monitor
1	8K x 8 Static RAMs or PROMs installed
2	8K x 8 Static RAMs or PROMs installed
3	8K x 8 Static RAMs or PROMs installed
4	User prototype may be RAM or PROM
5	User prototype may be RAM or PROM
6	All memory is on prototype, RAM or PROM
7	All memory above 7FFH is on prototype, RAM or PROM

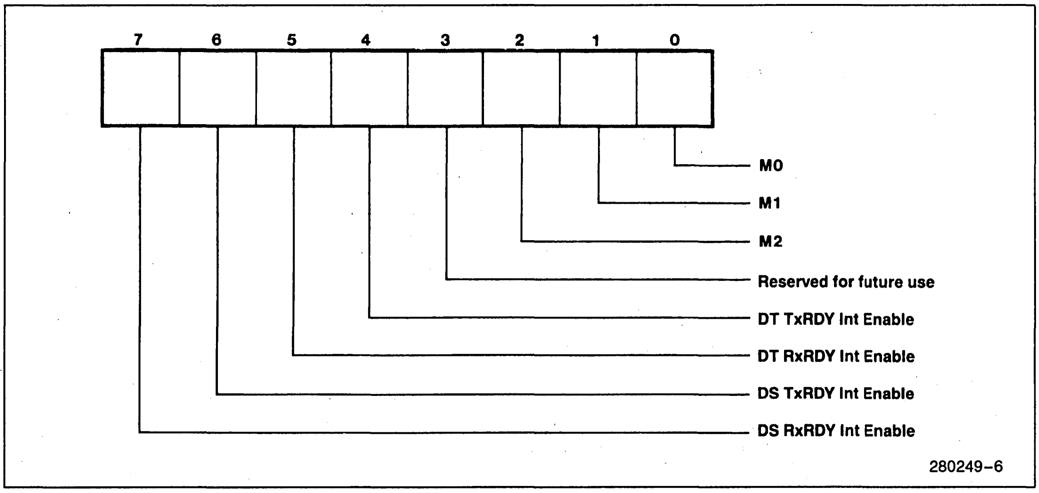


Figure 8. MODE Register Format

BREAKPOINTS

When emulation breakpoints or single-step emulation is used, the iSBE-96 monitor requires six bytes of the user's stack space. Since the ASM-96 assembler and the PL/M-96 compiler do not automatically take this into account, an extra six bytes of stack space needs to be allocated either explicitly in the code or implicitly with the STACKSIZE control of RL-96.

Since the trap vector (locations 2010H and 2011H) is utilized by the iSBE-96 emulator to provide breakpoints in emulation and single-step emulation, the trap vector locations must remain in RAM space or breakpoints and single stepping will not work. The iSBE-96 emulator could still go into emulation if these locations are in ROM or EPROM, but the ability to set breakpoints and single-step would be lost. In this case, emulation would be halted by sending an escape (<esc>) command to the iSBE-96 emulator.

When breakpoints are set, the instruction at the breakpoint is executed in single-step mode and not in real time. All other instructions up to the breakpoint are executed in real time. Here is one example of how the implementation of breakpoints affects debugging programs. Normally, a break on a PUSHF instruction at the start of a low priority interrupt service routine should enable the service routine to continue executing when emulation is resumed. Because the last instruction at the breakpoint is executed in non-real time, a higher priority interrupt could occur before the PUSHF instruction is actually executed. If this is the case, the higher priority interrupt would be serviced before the breakpoint at the PUSHF instruction. The breakpoint should be set on the instruction after the PUSHF if the higher priority interrupts need to be disabled.

MCS®-96 MICROCONTROLLER INTERRUPTS

All interrupt vector locations (2000H–200EH) should be initialized. This is a good practice even if the iSBE-96 emulator is not used for debug. This will prevent a system lock-up or crash in the event that the program unexpectedly enables interrupts. The vectors contain random addresses upon power up since the default memory map for the vector locations is in RAM. When a breakpoint is encountered during emulation, or while single-stepping, the monitor temporarily writes a trap instruction (0F7H) at all locations stored in the interrupt vectors. This could have adverse effects if the vector happened to contain the address of a register location, program data location or an instruction operand.

Any of the 8097 programmed events based on timer 1, timer 2 or external interrupts will continue to occur even while emulation of the iSBE-96 emulator has been stopped. When resuming emulation, these interrupts may be pending and would be serviced in order of priority. This could possibly cause an endless loop of service routines, overflow of the stack or differences between real-time emulation and emulation with breakpoints. Any code involving real-time events that has been debugged using breakpoints or single-step emulation should be verified in full speed, non-interrupted emulation.

Table 7. iSBE-96 Monitor Commands

Monitor Command	Function
BAUD	Sets up the baud rate.
BR	Enables display and setting of up to eight software breakpoints.
BYTE	Enables display and changing of a single byte or range of bytes of memory or a single byte of the 8097 internal registers.
CHANGE	Enables display and changing of a series of memory words or bytes.
<CONTROL>S	Stops scrolling of the screen display.
<CONTROL>Q	Resumes scrolling of the screen display.
<CONTROL>X	Deletes the line being entered.
<ESCAPE>	Aborts the command executing.
GO	Begins emulation and continues until an enabled breakpoint is reached or the escape key is pressed.
LOAD	Loads programs and data from disks.
MAP	Enables mapping of several preprogrammed memory maps; also enables configurable serial I/O and selective servicing of the watchdog timer.
PC	Displays and changes the program counter.
PSW	Displays and changes the program status word.
RESET CHIP	Resets the 8096 to power-up conditions.
SAVE	Saves programs and data to disks.
SP	Displays and changes the stack pointer.
STEP	Provides single-step emulation with selective display formats.
VERSION	Displays the monitor version number.
WORD	Enables display and changing of a single word or range of words of memory or a single word of the 8097 internal registers.

MCS®-96 Microcontroller Port 3/4

For anyone reconstructing port 3 and 4 (1FFEH and 1FFFH) on their target system, more care must be taken to debug the system. Since part of the port 3/4 reconstruction is an address decoder for 1FFEH and 1FFFH, the easiest thing to do is to temporarily change the mapped address for port 3/4 out of the reserved memory block. This means that both the hardware as well as the software has to be modified, but this enables debugging the integrated hardware and software. The software could automatically change the port addresses for debugging with the use of conditional assemble or compile statements.

The other method for debugging port 3/4 requires that the hardware and software be debugged separately or at least in stages. The user system, except for the port 3/4 reconstruction and any code utilizing port 3/4, would have to be debugged first. Then, with the iSBE-96 emulator in port 3/4 configuration (using MAP BUSPINS = PORT 34), the iSBE-96 emulator would be connected directly to the user's system port 3/4 pins. That is, the iSBE-96 port 3/4 pins on connector J4 would be connected on the port side of the user's port 3/4 reconstruction, bypassing it altogether.

CONNECTING THE iSBE-96 EMULATOR TO THE IBM PC XT AND AT

Introduction

A communications program (driver) is supplied with the iSBE-96 emulator so that it can be used with an

IBM PC XT and AT, as well as an Intel Series III or Series IV development system. This driver provides an enhanced command set (extensions shown in Table 8) for the iSBE-96 emulator and provides access to the host system's mass storage.

The following sections describe the additional features provided by the driver.

iSBE-96 Emulator Additional Commands Available

In addition to the command set provided by the iSBE-96 monitor, the driver provides a set of computer system interface commands. The additional commands provided by the driver are summarized in Table 8. The driver provides the proper communications protocol to complete the implementation of the iSBE-96 monitor LOAD and SAVE command. The LIST command will save a copy of everything displayed on the console to a system file, creating a complete log of the emulation session for future reference. Also, the INCLUDE command will redirect command input to come from a system file.

iSBE-96 Emulator Symbolic Support

The iSBE-96 monitor supports the use of symbolics for the program counter (PC), program status word (PSW), and stack pointer (SP). Additionally, the driver supports symbolics for the MCS-96 special function registers in the ASM and DASM commands. With this feature, the symbolic reference can be to a special function register when using the ASM and DASM commands rather than the register address, which can be cumbersome to remember or look up. Figure 9 contains a list of the symbolics supported by the ASM and DASM commands. These symbols are compatible with the MCS-96 symbols listed in Figure 4.

MODIFYING THE iSBE-96 EMULATOR CLOCK SPEED

Introduction

Although it comes standard with a 12 MHz crystal, the iSBE-96 emulator is designed to operate at crystal frequencies from 6 MHz to 12 MHz. The iSBE-96 monitor power-up diagnostics include board-level serial port tests that take advantage of the 12 MHz crystal frequency. Therefore, to operate the iSBE-96 emulator at other crystal frequencies, it is necessary to disable the power-up diagnostics. Only two simple modifications are needed: altering the monitor code and changing the crystal itself.

iSBE-96 Monitor Patch

The first modification disables the power-up diagnostics. This is completed by changing the monitor's 3-byte CALL instruction to the diagnostics to NOP (no-operation) instructions. The call to diagnostics is located at

Table 8. Driver Commands

Driver Command	Function
ASM	Loads memory with translated MCS-96 assembler mnemonics.
DASM	Displays memory as MCS-96 assembler mnemonics.
EXIT	Exits the driver and returns to the host operating system.
<CONTROL>C	Same as for the EXIT command, but will not properly close the system serial port.
HELP	Displays the syntax of all commands.
INCLUDE	Specifies a command file.
<CONTROL>I	Turns the command file on and off.
<TAB>	Same as <CONTROL>I (turns the command file on and off).
LIST	Specifies a list file.
<CONTROL>L	Turns list file on and off.
<CONTROL>S	Stops scrolling of the screen display.
<CONTROL>Q	Resumes scrolling of the screen display.
<CONTROL>X	Deletes the line being entered.
<ESCAPE>	Aborts the command executing.

EPROM address 1046H (monitor address 208CH). The following is a step-by-step explanation of what to do to the monitor, version 1.1, to make the patch.

1. Remove the low-byte monitor EPROM (U53) and, using a PROM programmer, copy its contents to the PROM programmer data buffer.
2. Change bytes 1046H and 1047H in the data buffer from 0EFH and 32H, respectively, to 0FDH.
3. Using another 27128 EPROM with 250 nanosecond access time, program a new monitor PROM and install it in the iSBE-96 emulator as U53.
4. Remove the high-byte monitor EPROM (U61) and, using a PROM programmer, copy its contents to the PROM programmer data buffer.
5. Change byte 1046H in the data buffer from 2CH to 0FDH.
6. Using another 27128 EPROM with 250 nanosecond access time, program a new monitor PROM and install it in the iSBE-96 emulator as U61.

With this change in place the DIAGS LED on the iSBE-96 emulator will not go off after power-up. If for any reason you suspect a problem with the iSBE-96 emulator, reinstall the original monitor PROMs and use the power-up diagnostics for system checkout or before servicing the iSBE-96 emulator.

iSBE-96 Crystal Modification

There are now two ways to modify the iSBE-96 emulator to operate at different clock speeds. The first is by far easier and the second involves more work.

The first method of modifying the iSBE-96 emulator is to simply replace the 12 MHz crystal, Y1, with the desired crystal. The only restriction is that the new crystal must be between 6 MHz and 12 MHz.

The second method is to modify the iSBE-96 emulator to use the target system crystal frequency. To do this, carefully remove crystal Y1 and capacitors C6 and C7 from the iSBE-96 emulator. The target system crystal oscillator should be buffered with the circuit shown in Figure 10. The buffer output connects to the empty Y1 board connection closest to the edge of the board, as shown in Figure 11. The target system clock is also limited to 6 MHz through 12 MHz.

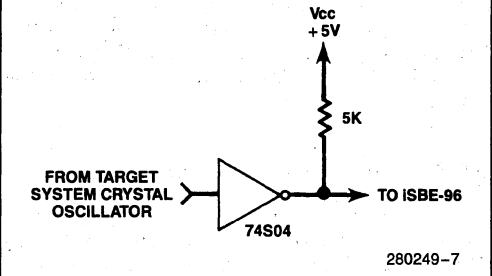


Figure 10. External Clock Drive

AD_COMMAND	IOPORT2
AD_RESULT	IOPORT3
AD_RESULT_HI	IOPORT4
AD_RESULT_LO	IOS0
BAUDRATE	IOS1
HSI_MODE	PWM_CONTROL
HSI_STATUS	SBUFRX
HSI_TIME	SBUFTX
HSI_TIME_HI	SP
HSI_TIME_LO	SP_CONN
HSO_COMMAND	SP_STAT
HSO_TIME	TIMER1
HSO_TIME_HI	TIMER1_HI
HSO_TIME_LO	TIMER1_LO
INT_MASK	TIMER2
INT_PENDING	TIMER2_HI
IOC0	TIMER2_LO
IOC1	WATCHDOG
IOPORT0	ZERO
IOPORT1	

Figure 9. ASM and DASM Command Symbol Support List

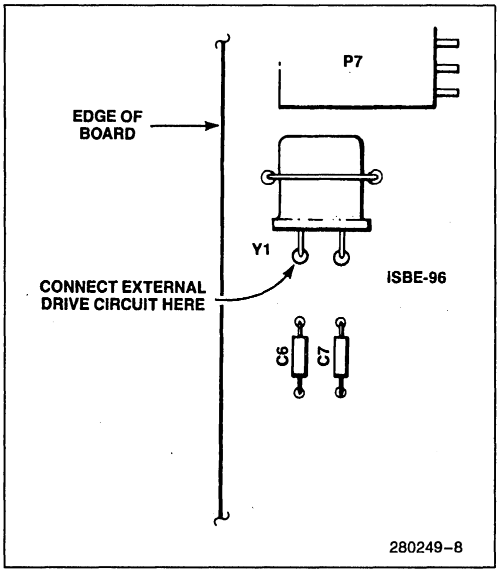


Figure 11. External Clock Connection

Care should be taken to ensure adequate digital ground connections between the target system and the iSBE-96 emulator. The user cable connected to J4 can be used for that purpose. All even numbered pins on J4 (except for pin 2) are connected to digital ground on the iSBE-96 emulator.

Before having your iSBE-96 emulator serviced by Intel, it should be restored to its original condition.

MODIFYING THE iSBE-96 MEMORY MAP

Introduction

The iSBE-96 emulator provides seven user memory map (mode) selections. There are eight total, but the monitor reserves the use of one map, mode zero. The iSBE-96 memory maps are illustrated in Figure 7. Even though these memory maps fulfill the majority of the user's needs, there will be times when a custom memory map is desired. This can be done easily if you follow the guidelines in this section.

The memory space for the MCS-96 microcontroller, as well as the 8097 used on the iSBE-96 emulator, has a range from 0 to 64K (0FFFFH) bytes. The 8097 has a linear memory space, but the data bus from the off-chip memory's even bytes are connected to the low eight data pins of the 8097 and the odd bytes are connected to the upper eight data pins. Therefore, if the memory map needs to be changed, it should be changed along even byte boundaries (2K, 4K, 16K, 32K) and should account for pairs of byte-wide memory chips (i.e., 2-2K x 8 and 2-8K x 8).

There are only two blocks of memory that have restrictions on them with the iSBE-96 emulator. These blocks are locations 0 through 0FFH and 1F00H through 1FFFH. These blocks are reserved for use by the iSBE-96 emulator and should always be mapped accordingly.

iSBE-96 Memory Map PROM

Before changing the iSBE-96 memory map PROM, it will help to know what it is and what it does.

The iSBE-96 memory map PROM (U39) is a 2K x 8 bipolar PROM. Since PROMs are one-time programmable, chances are that any changes will require replacement of the PROM. There is one key parameter when finding a replacement for the iSBE-96 memory map PROM, the time required from valid address on the input pins of the PROM to valid data on the output pins (t_{avdv}). The iSBE-96 memory map PROM requires a t_{avdv} time of 35 nanoseconds or better. An Intel 3636B-1 or any PROM satisfying the time requirements and having the standard JEDEC pin configuration can be used. Figure 12 shows the pin out and functional connections of the iSBE-96 memory map PROM.

Since the iSBE-96 memory map PROM is 2K bytes and there are eight memory maps, the memory map PROM is functionally segmented into eight blocks of 256 bytes each. Figure 13 illustrates the map PROM block assignments. Each block contains the map for one of the eight iSBE-96 monitor memory maps (modes) and each byte within a block contains the 'map' for 256 bytes of the total 64K byte address range. Figure 14 shows what the map byte contents should be to enable the different memory areas that are re-mappable.

The DATARAM (locations 100H through 7FFH) is not totally re-mappable. The DATARAM can be relocated to any 4K area in the 64K address range, but it always has to be at locations 100H through 7FFH in that 4K area.

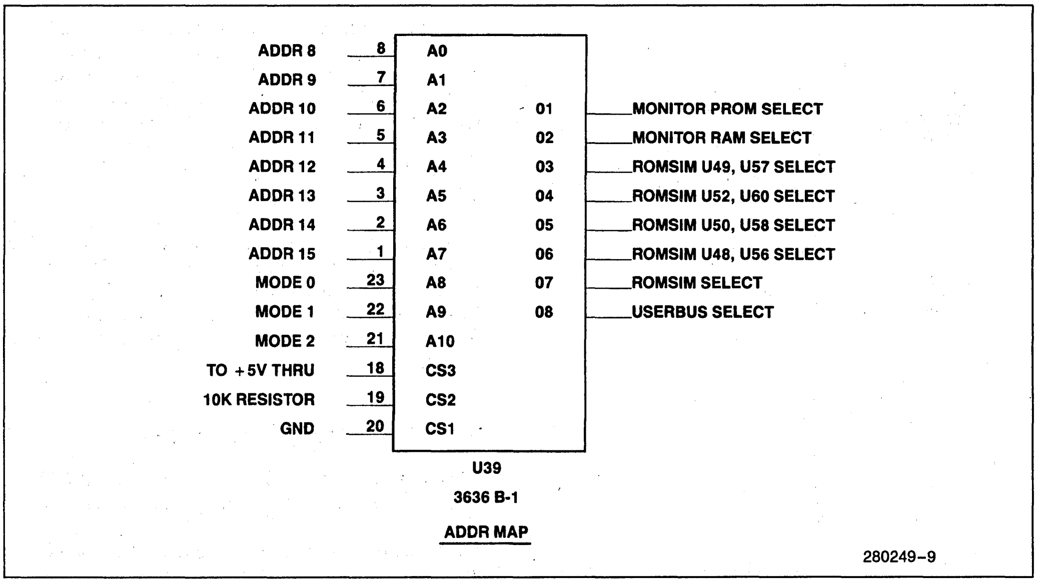


Figure 12. iSBE-96 Address Map PROM

MAP PROM Address	Monitor Memory Mode
0–0FFH	0
100–1FFH	1
200–2FFH	2
300–3FFH	3
400–4FFH	4
500–5FFH	5
600–6FFH	6
700–7FFH	7

Figure 13. MAP PROM Blocks

Chip Location	MAP Byte	Current MAPPED Address
U49–U57	0BBH	2000–2FFFH
U52–U60	0B7H	3000–3FFFH
U50–U58	0AFH	4000–4FFFH
U48–U56	9FH	5000–5FFFH
User	7FH	—
DATA RAM	0BDH	100–7FFH

Figure 14. iSBE-96 Map PROM Key

Sample iSBE-96 Memory Map Modification

As an example, let's say I have an iSBE-96 memory map that matches the map of the system I am developing. The map that I want needs to have locations 100H through 10FFH for mapped I/O devices, 1100H through 17FFH for scratch pad RAM, and 2000H through 0FFFFH for my EPROM application.

The I/O in my system is working, but I don't have the scratch pad RAM working yet and I don't want to program EPROMs until I have debugged my application program. So, what I really want is the scratch pad RAM mapped to iSBE-96 DATARAM and my EPROM memory area mapped to iSBE-96 RAM (ROMSIM). To accomplish the mapping for the EPROM, the iSBE-96 ROMSIM will have to be replaced by larger RAMs, as shown in Figure 15.

After looking at the different map modes (see Figure 7) I can see that mode 2 is close, but not quite it. So, mode 2 is the mode that I decide to change.

The following are the steps necessary to make the change.

1. Remove U48–U50, U52, U56–U58, and U60 on the iSBE-96 emulator.
2. Install 8K x 8, 150 nanosecond t_{avdy} static RAMs in their place and jumper the iSBE-96 emulator per Table B-2 in the iSBE-96 User's Guide, shown here as Table 9.
3. Remove the iSBE-96 map PROM (U39) and, using a PROM programmer, copy its contents to the PROM programmer data buffer.
4. Change bytes 100H through 10FFH to 7FH, and 1100H through 17FFH to 0BDH.
5. Program a new map PROM and install it as U39.

The new memory map could now be accessed by entering MAP MODE = 2 on the iSBE-96 console.

As you did with the monitor PROMs, the original address map PROM should be retained in case the iSBE-96 needs to be serviced by Intel.

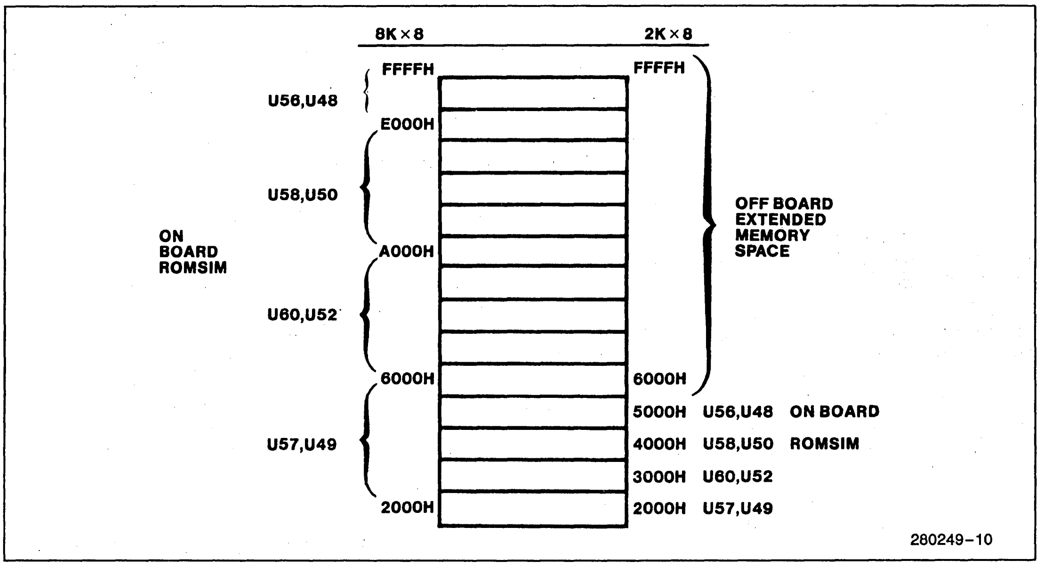


Figure 15. 8K x 8 Address Map

Table 9. 8K x 8 Replacement Jumper Configuration

Jumper Change		Function Incorporated by the Change
Default	Replacement	
E13–E14	E14–E15	Connects MA12 to U48
E16–E17	E17–E18	Connects MA12 to U49
E22–E23	E23–E24	Connects MA12 to U50
E31–E32	E32–E33	Connects MA12 to U52
E39–E40	E40–E41	Connects MA12 to U56
E47–E48	E48–E49	Connects MA12 to U57
E58–E59	E59–E60	Connects MA12 to U58
E77–E78	E78–E79	Connects MA12 to U60
E19–E20	OPEN	Disconnects U49, U57 pin 26 from VCC(1)
E36–E37	OPEN	Disconnects U48, U56 pin 26 from VCC(1)
E55–E56	OPEN	Disconnects U50, U58 pin 26 from VCC(1)
E74–E75	OPEN	Disconnects U52, U60 pin 26 from VCC(1)

NOTE:
1. It may be desirable to leave pin 26 connected to V_{CC}. Check pin out for 8K x 8 device used.

HELPFUL MCS®-96 PROGRAMS FOR THE iSBE-96

Introduction

During operation we discovered that the iSBE-96 emulator would be even more useful if it had a few more, or slightly different, commands. The following sections describe some helpful MCS-96 programs that can be used on the iSBE-96 emulator to make debugging your programs a little easier.

Memory Write Without Read Verify

As you may have already discovered, the iSBE-96 BYTE, WORD, and CHANGE commands do a read verify after writing the specified memory locations. This is very useful for determining if the memory is functioning, but requires that the memory be RAM. What then do you do if your system has memory mapped peripheral devices that access different registers for a read and write operation? The BYTE and WORD commands will write the location(s) correctly, but they will display a read verify error message.

Figure 16 illustrates an ASM-96 program that will perform the write to memory without a read verify. The program is located at 100H to correspond to the iSBE-96 DataRAM and thereby not intrude into user memory space. The program also uses eight bytes of internal 8097 register space. Again, so that the program does not intrude, the eight register bytes are pushed onto the stack and restored upon exit. You will have to ensure that there is sufficient stack available. The data structure containing the bytes and their respective addresses is assumed to be structured as follows: 150H byte containing the count of data bytes, 152H first data byte, 152H + byte count (+1 if byte count is odd) address for first data byte.

To use the program, first make sure you are in an iSBE-96 memory mode that provides DataRAM, then load the program object code. Once the program is loaded, put the data into the data structure at 150H: byte count, data bytes followed by data addresses. To execute the program simply type "GO FROM 100 TO 140". When the program stops at the breakpoint, the data bytes will have been written to the specified addresses.

Block Memory Move

If you have ever put something into memory and then decided that it should be located at another address, then you've probably wanted a block move program. It becomes tedious to move data structures or code a byte or word at a time. Sometimes it is inconvenient to relocate or link the original object code so that it can be loaded at the new location.

Since the MCS-96 instruction set utilizes relative offsets for the majority of the jump and branch instructions, it is feasible to move code blocks around. Of course, the block of code that you intend to move has to be either self-contained or small enough to fit within that mode of addressing. That is, the block of code moved should not contain a relative jump or branch to anywhere outside the block.

Figure 17 illustrates an ASM-96 program that will perform a block memory move. The program is located at 200H to correspond with the iSBE-96 DataRAM and so that it will not interfere with the write program described previously in "Memory Write Without Read Verify" section which is located at 100H. The program uses eight bytes of internal 8097 register space. So that the program is nonintrusive, the eight register bytes are pushed onto the stack and restored upon exit. You will have to ensure that there is sufficient stack available. The data structure containing the start, stop and destination addresses is assumed to be structured as follows: 230H start address, 232H stop address, and 234H destination address.

To use the program, first make sure you are in an iSBE-96 memory mode that provides DataRAM, then load the program object code. Once the program is loaded, put the data into the data structure at 230H: start address, stop address and finally destination address. To execute the program simply type "GO FROM 200 TO 22C". When the program stops at the breakpoint, the block of memory will have been moved to the specified location.

Writing/Reading an iSBE-96 Terminal in Emulation

There may be times while a program is executing that you would like to know how far it has progressed. But, you may not wish to use breakpoints to check the progress because they change the overall execution speed. This is particularly true for programs using real-time interrupts, since it may not be possible to use breakpoints. Since the iSBE-96 serial ports (DCE and DTE) are accessible during emulation, you can include program routines that write to a terminal or from the terminal to relay program status or dynamically change the program flow, provided you do it with care.

The iSBE-96 emulator uses the on-board 8097 NMI interrupt to service the DCE and DTE serial ports. This occurs even in emulation since there are some commands that are valid during emulation. Therefore, care should be taken when utilizing the unused serial port for dynamic program status. Since the iSBE-96 emulator is always connected to the host development system via the DCE serial port, a terminal can be connected to the unused DTE serial port. Incidentally, if you want to see what you're typing your program will need to echo it to the terminal.

```
MCS-96 MACRO ASSEMBLER        8096 Write with no Read Verify Routine        01/14/86

DOS MCS-96 MACRO ASSEMBLER, V1.0

SOURCE FILE: WRITE.A96
OBJECT FILE: WRITE.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: <none>

ERR LOC  OBJECT      LINE         SOURCE STATEMENT
                        1   $TITLE ('8096 Write with no Read Verify Routine')
                        2
                        3
                        4   Write   MODULE  MAIN
                        5
    0100                6           CSEG at 100h
                        7
    0100 C820           8   start:  push    20h             ;save working registers
    0102 C822           9           push    22h
    0104 C824          10           push    24h
    0106 C826          11           push    26h
    0108 B301500120    12           ldb     20h,150h        ;load byte count
    010D 990020        13           cmpb    20h,#0          ;make sure there are
                                                            ;bytes to write
    0110 DF26          14           je      J3
    0112 B10021        15           ldb     21h,#0          ;initialize registers
    0115 A1520122      16           ld      22h,#152h
    0119 C02420        17           st      20h,24h
    011C 302004        18           jbc     20h,0,J1        ;see if byte count is odd
    011F 65010024      19           add     24h,#1          ; if odd, add 1 for even
                                                            ;boundary
    0123 65520124      20   J1:     add     24h,#152h       ;load location of first byte
                                                            ;address
    0127 A22426        21   J2:     ld      26h,[24h]       ;load data byte address
    012A B22321        22           ldb     21h,[22h]+      ;load data byte and
                                                            ;increment pointer
    012D C62621        23           stb     21h,[26h]       ;write the byte
    0130 65020024      24           add     24h,#2          ;increment pointer to next
                                                            ;address
    0134 1520          25           decb    20h             ;done yet?
    0136 D2EF          26           jgt     J2
    0138 CC26          27   J3:     pop     26h             ;restore working registers
    013A CC24          28           pop     24h
    013C CC22          29           pop     22h
    013E CC20          30           pop     20h
    0140 27FE          31   J4:     br      J4              ;wait here
                       32
    0142               33   END
                                                                         280249-11
```

Figure 16

```
MCS-96 MACRO ASSEMBLER     8096 Block Memory MOVE Routine                    01/14/86

DOS MCS-96 MACRO ASSEMBLER, V1.0

SOURCE FILE: MOVE.A96
OBJECT FILE: MOVE.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: <none>

ERR  LOC   OBJECT       LINE   SOURCE STATEMENT
                           1   $TITLE ('8096 Block Memory MOVE Routine')
                           2
                           3
                           4   Move    MODULE   MAIN
                           5
     0200                  6           CSEG at 200h
                           7
     0200  C820            8   start:  push     20h          ;save working registers
     0202  C822            9           push     22h
     0204  C824           10           push     24h
     0206  C826           11           push     26h
     0208  A301300220     12           ld       20h,230h     ;load start address
     020D  A301320222     13           ld       22h,232h     ;load end address
     0212  A301340224     14           ld       24h,234h     ;load destination address
     0217  882022         15   J1:     cmp      22h,20h      ;make sure there is
                                                             ;something to move
     021A  DE08           16           jlt      J2           ;if equal then only one
                                                             ;byte to move
     021C  B22126         17           ldb      26h,[20h]+   ;load byte and increment
                                                             ;source pointer
     021F  C62526         18           stb      26h,[24h]+   ;store byte and increment
                                                             ;destination pointer
     0222  27F3           19           br       J1           ;go see if done
     0224  CC26           20   J2:     pop      26h          ;restore working registers
     0226  CC24           21           pop      24h
     0228  CC22           22           pop      22h
     022A  CC20           23           pop      20h
     022C  27FE           24   J3:     br       J3           ;wait here
                          25
     022E                 26   END

SYMBOL TABLE LISTING
--------------------

N A M E                                VALUE    ATTRIBUTES
J1. . . . . . . . . . . . . . .        0217H    CODE ABS ENTRY
J2. . . . . . . . . . . . . . .        0224H    CODE ABS ENTRY
J3. . . . . . . . . . . . . . .        022CH    CODE ABS ENTRY
MOVE. . . . . . . . . . . . . .        -----    MODULE MAIN STACKSIZE(0)
START. . . . . . . . . . . . .         0200H    CODE ABS ENTRY

ASSEMBLY COMPLETED,        NO ERROR(S) FOUND.
                                                                      280249-12
```

Figure 17

Figure 18 illustrates the PL/M-96 procedures to read and write a terminal connected to the DTE serial port on the iSBE-96 emulator and a sample calling program. The sample program uses an initial delay to ensure that the iSBE-96 NMI line has stabilized so that spurious NMI interrupts are not caused by accessing the DTE serial port. Figure 19 illustrates steps to compile and link the sample program.

To run the program, first load the sample program object code into the iSBE-96 emulator using the LOAD command. Then, type "GO FROM 2080 FOREVER". When you are ready to stop, press the escape key and emulation will halt.

iSBE-96 SERIAL PROTOCOL FOR LOAD AND SAVE

Introduction

The iSBE-96 emulator has a number of resident monitor commands, as described in Table 7. Normally, the iSBE-96 emulator is hosted by an IBM PC XT or AT, or an Intel Series III or Series IV development system. If you have a different host, you must write your own software program (driver) that meets the software handshaking protocol required by the iSBE-96 emulator. In that way, the resident monitor commands can be executed with any computer or terminal connected to the iSBE-96 DCE or the DTE serial ports.

The normal configuration is for the iSBE-96 emulator to be attached to a host computer system on the DCE port. Alternately, the iSBE-96 emulator can be attached to a terminal on the DTE port, which leaves the DCE port free to be connected to a computer. The terminal would be used to enter iSBE-96 debug commands (including LOAD and SAVE) and the computer used solely for loading and saving MCS-96 program files.

Whichever way you do it, the proper iSBE-96 serial port (DCE,DTE) needs to be mapped appropriately for loading, saving, and console connections. The MAP CONSOLE command is used to change the serial port connection for the console device. The iSBE-96 emulator will default to the port that the console is connected to at power up. The MAP SEND command is used to designate which serial port the iSBE-96 monitor uses for data transfer (sending) for the SAVE command. The MAP RECEIVE command is used to designate which serial port the iSBE-96 monitor uses for data transfer (receiving) for the LOAD command.

The next three sections describe the handshaking protocol used by the iSBE-96 emulator for loading and saving files. They provide sufficient information to write your own program to load and save programs with the iSBE-96 emulator.

Handshaking Characters

There are two characters that are used for control during the actual file transfer, EOF (1AH) and ESC (1BH). Determination that one of the two control characters has been encountered requires the use of a third character, DLE (10H). When transferred as data, the DLE, EOF and ESC characters must be prefixed by a DLE character. Additionally any data byte when ANDed with 7FH that yields one of the control characters (90H,9AH, and 9BH) also needs to be prefixed by a DLE. DLEs sent as prefixes should not be included in the byte count and should not be stored as data.

Loading Files

The following describes the protocol required by the iSBE-96 emulator for loading files. The following terminology is used: <cr> denotes a carriage return; Console is the terminal or computer mapped to the iSBE-96 CONSOLE device; Sender is the computer mapped to the iSBE-96 SEND device; Receiver is the computer mapped to the iSBE-96 RECEIVE device.

1. Console sends 'LOAD <cr>' to the iSBE-96.
2. iSBE-96 sends an XON (11H) to Console.
3. Sender sends up to 16,384 bytes and waits for iSBE-96 to send an XON (11H).
4. iSBE-96 processes the transferred bytes and sends an XON (11H) to Sender.
5. Steps 3 and 4 are repeated until the transfer is complete.
6. Sender sends an EOF (1AH) to iSBE-96.
7. iSBE-96 sends a prompt ("*") to Console.

If, during the transfer, the iSBE-96 emulator receives an unprefixed ESC (1BH) from the Sender or from the Console, the load is aborted and an ESC is sent to the Sender. The Sender should then respond with an XON (11H) to acknowledge the ESC.

If the end of file is reached at any time during the load, the transfer is terminated. The full 16,384 (16KH) bytes do not necessarily have to be transferred.

```
DOS PL/M-96 V1.0 COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN TERMRW.OBJ
COMPILER INVOKED BY: C:\UDI\PLM96.EXE TERMRW.P96
                 $title (' iSBE-96 Terminal Read/Write Sample Program')
                 $optimize (3)
   1             sample: DO;
                 /* local declarations */
   2    1        DECLARE  msg1(*)    BYTE   DATA(44H,61H,76H,65H,20H,53H,63H,68H,
                                                 6FH,65H,62H,65H,6CH,20H,69H,73H,
                                                 20H,47H,52H,45H,41H,54H),
                          msg2(*)    BYTE   DATA(0dH,0aH),
                          msg3(*)    BYTE   DATA(72H,69H,67H,68H,74H,3FH),
                          (I,char)   BYTE,
                          dt_data    ADDRESS AT (1FE4H),
                          dt_status  ADDRESS AT (1FE6H),
                          bell       LITERALLY    '07H';
                 /* Procedure declarations */
   3    1        ci: PROCEDURE   BYTE   PUBLIC;
   4    2        DO WHILE ((dt_status AND 02H) = 0H);      /* wait till RxRDY */
   5    3            END;
   6    2        char = dt_data AND 7FH;
   7    2        RETURN char;
   8    2        END ci;
   9    1        co: PROCEDURE (char)      PUBLIC;
  10    2        DECLARE char      BYTE;
  11    2        DO WHILE ((dt_status AND 1) = 0 );        /* wait till TxRDY */
  12    3            END;
  13    2        dt_data = char;
  14    2        END co;
                 /* Program starts here */
  15    1        CALL TIME(50);
  16    1        dt_status=37H;                            /* clear any errors on the DTs 8251A
                                                           USART */
  17    1        CALL TIME(1);
  18    1        char = dt_data;                           /* clear the DT receive buffer */
  19    1        DO I = 1 TO LENGTH(msg1);
  20    2            CALL co(msg1(I-1));
  21    2            END;
  22    1        char = 'n';
  23    1        DO WHILE (char = 'n');
  24    2            DO I = 1 TO LENGTH(msg2);
  25    3              CALL co(msg2(I-1));
  26    3              END;
                                                                              280249-13
```

Figure 18

```
  27      2        DO I = 1 TO LENGTH(msg3);
  28      3          CALL co(msg3(I-1));
  29      3          END;
  30      2        char = ci;
  31      2        CALL co(char);
  32      2        IF ((char< >'Y') AND (char< >'y')) THEN DO;
  34      3          char = 'n';
  35      3          CALL co(bell);
  36      3          END;
  37      2        END;

  38      1    DO WHILE 1;                              /* wait here when done */
  39      2        END;

  40      1    END sample;

MODULE INFORMATION:

  CODE AREA SIZE                    = 00DBH        219D
  CONSTANT AREA SIZE                = 001EH         30D
  DATA AREA SIZE                    = 0000H          0D
  STATIC REGS AREA SIZE             = 0003H          3D
  OVERLAYABLE REGS AREA SIZE        = 0000H          0D
  MAXIMUM STACK SIZE                = 0004H          4D
  60 LINES READ

PL/M-96 COMPILATION COMPLETE.     0 WARNINGS,     0 ERRORS
```

280249–14

Figure 18 (Continued)

```
  C:\SBE96 >plm96 termrw.p96

DOS PL/M-96 COMPILER V1.0
Copyright Intel Corporation 1983
PL/M-96 COMPILATION COMPLETE.     0 WARNINGS,     0 ERRORS

  C:\SBE96 >rl96 termrw.obj,plm96.lib to termrw.abs stacksize(16)

DOS MCS-96 RELOCATOR AND LINKER, V2.0
Copyright 1983 Intel Corporation
RL96 COMPLETED,          0 WARNING(S),          0 ERROR(S)

  C:\SBE96 >
```

280249–15

Figure 19

Saving Files

The following describes the protocol required by the iSBE-96 emulator for saving files. The following terminology is used: <cr> denotes a carriage return; partition denotes an address range, specified as 'address TO address'; Console is the terminal or computer mapped to the iSBE-96 CONSOLE device; Sender is the computer mapped to the iSBE-96 SEND device; Receiver is the computer mapped to the iSBE-96 RECEIVE device.

1. Console sends 'SAVE partition <cr>' to iSBE-96.
2. iSBE-96 sends an STX (02H) to Receiver.
3. Receiver acknowledges with an XON (11H) to iSBE-96.
4. iSBE-96 sends up to 16,384 bytes and waits for Receiver to send an XON (11H).
5. Receiver processes the transferred bytes and sends an XON (11H) to the iSBE-96.
6. Steps 4 and 5 are repeated until the transfer is complete.
7. iSBE-96 sends an EOF (1AH) to Receiver.
8. iSBE-96 sends a prompt ("*") to Console.

If, during the transfer, the iSBE-96 emulator receives an ESC (1BH) from the Receiver or from the Console, the load is aborted and an ESC is sent to the Receiver. The Receiver should then respond with an XON (11H) to acknowledge the ESC.

If the end of file is reached at any time during the load, the transfer is terminated. The full 16,384 (16KH) bytes do not necessarily have to be transferred.

SAMPLE DEBUG SESSION WITH THE iSBE-96 EMULATOR

The following sample program requires the use of PL/M-96, ASM-96, and an iSBE-96 emulator. It assumes the iSBE-96 DCE serial port is connected to an IBM PC XT or AT and a terminal is connected to the iSBE-96 DTE serial port. The terminal should be set for full-duplex and 9600 baud operation.

Sample Program Description

The MCS-96 program chosen for the sample debug session combines and utilizes many of the features described throughout this applications note and was designed to show as many of the iSBE-96 emulator's features as possible. The sample program uses both a PL/M-96 main module and an ASM-96 module and demonstrates how to link them together. The sample program also uses the terminal input/output procedures discussed in the Block Memory Move Section for input to the program and to display status in real-time. Finally, the program makes use of one of the MCS-96 software timers for basic program timing.

The PL/M-96 main module is illustrated in Figure 20. As shown, the main module contains local declarations, procedure declarations, and the mainline PL/M-96 program. Functionally, the program uses software timer 1 to keep a real time clock which is then displayed to the terminal connected to the iSBE-96 DT serial port. Initially the 'clock' is set by entering the current time through the terminal connected to the iSBE-96 DT port.

The ASM-96 module is shown in Figure 21. It contains the interrupt service routine for the software timer interrupt which actually does the timing for the 'clock'. It also defines all of the other MCS-96 interrupt vectors (2000H to 200FH) to help guard against program runaway and to avoid program anomolies when debugging with the iSBE-96 emulator.

Figure 22 illustrates the DOS batch file (CLOCK.BAT) used to compile, assemble, and link the sample program. The STACKSIZE(20H) control is added to the RL96 invocation to allow sufficient stack space for the sample program and the six bytes required by the iSBE-96 emulator. This batch file assumes that PL/M-96, ASM-96 and the utilities and libraries are located in a directory called 8096.DIR while the sample program modules and batch file are in the home directory. After entering the sample program modules and batch file using a word processor such as AEDIT, the sample program can then be assembled, compiled, and linked by typing CLOCK followed by an enter.

If a word processor other than AEDIT is used, you should insure that the word processor did not put an end of file character (IAH) at the end of the source code files since the Intel assemblers and compilers cannot handle it. It can be removed using the DOS copy/b command.

Sample Program Discussion

Before beginning the sample debug session it may be helpful to have a brief synopsis of what the sample program does and why. The MCS-96 software timers are incremented once every eight state times and the maximum count possible for the software timers is 65,535 (64KH). For a 12 MHz input crystal frequency, a state time is 250 ns. Therefore, one second can be expressed as: 1 = 1/(250E−9 * 8 * 65,535 * X) where X is the number of times the software timer completes the specified number of counts (time-outs). If you solve for X you will find that X = 7.6295. This tells us that we need seven time-outs at the maximum count and one time-out at a count of 41,254 (65,535 * 0.6295).

```
PL/M-96 COMPILER          iSBE-96 Sample Debug Program
DOS PL/M-96 V1.0 COMPILATION OF MODULE CLOCK
OBJECT MODULE PLACED IN CLOCK.OBJ
COMPILER INVOKED BY:     C:\UDI\PLM96.EXE CLOCK.P96
                 $title (' iSBE-96 Sample Debug Program')
                 $optimize (3)
  1              clock: DO;
                 /* local declarations */
  2    1         DECLARE  bell          LITERALLY   '07H',
                          BS            LITERALLY   '08H',
                          FOREVER       LITERALLY   'WHILE 1',
                          FALSE         LITERALLY   '0',
                          TRUE          LITERALLY   'NOT FALSE',
                          BOOLEAN       LITERALLY   'BYTE',
                          msg1(*)       BYTE        DATA(0dH,0aH),
                          msg2a(*)      BYTE        DATA(0,0,':',0,0,':',0,0),
                          msg2(8)       BYTE        FAST,
                          msg3(*)       BYTE        DATA('set time - hh:mm:ss <cr>'),
                          (I,char)      BYTE,
                          seconds       BYTE,
                          minutes       BYTE,
                          hours         BYTE,
                          tick          BYTE        FAST    PUBLIC,
                          tock          WORD        PUBLIC,
                          count         BYTE        EXTERNAL,
                          count1        BYTE,
                          not$done      BOOLEAN,
                          not$first     BOOLEAN,
                          HSO_TIME      WORD        AT (04H),
                          HSO_CMD       BYTE        AT (06H),
                          INT_MASK      BYTE        AT (08H),
                          INT_PENDING   BYTE        AT (09H),
                          TIMER1        WORD        AT (0AH),
                          dt_data       ADDRESS     AT (1FE4H),
                          dt_status     ADDRESS     AT (1FE6H);
                 /* Procedure declarations */
  3    1         ci: PROCEDURE     BYTE     PUBLIC;
  4    2         DO WHILE ((dt_status AND 02H) = 0H);          /* wait till RxRDY */
  5    3             END;
  6    2         char = dt_data AND 7FH;
  7    2         RETURN char;
  8    2         END     ci;
  9    1         co:     PROCEDURE (char)     PUBLIC;
 10    2         DECLARE     char     BYTE;
 11    2         DO WHILE ((dt_status AND 1) = 0 );            /* wait till TxRDY */
 12    3             END;
```

280249–16

Figure 20

```
2   dt_data = char;
2   END      co;
1   init$DT:    PROCEDURE    PUBLIC;
2   dt_status = 37H;                                    /* clear any errors on the DTs
                                                        8251A USART */
2   CALL TIME(1);
2   char = dt_data;                                     /* clear the DT receive buffer */
2   END      init$DT;
1   ascii:    PROCEDURE (value,dest$ptr)    PUBLIC;
2   DECLARE (value,temp)     BYTE,
            dest$ptr     ADDRESS,
            (dest     BASED     dest$ptr) (2)     BYTE;
2   value = SHL((value/10),4) + (value MOD 10);         /* convert to BCD */
2   temp = value;
2   dest(0) = SHR(temp,4) + 30H;                        /* convert to ASCII decimal value */
2   dest(1) = (value AND 0FH) + 30H;
2   END      ascii;
1   print$msg1:     PROCEDURE;
2   DECLARE     I     BYTE;
2   DO I = 1 TO LENGTH(msg1);
3       CALL co(msg1(I-1));
3       END;
2   END      print$msg1;
    /* Program starts here */
1   CALL TIME(50);                                      /* delay to insure iSBE-96 NMI line
                                                        is stable */
1   CALL init$DT;                                       /* initialize DT serial port */
1   count,count1 = 0;                                   /* initialize variables */
1   not$done = TRUE;
1   not$first,tick = FALSE;
1   seconds,minutes,hours = 0;
1   CALL movb(.msg2a,.msg2,LENGTH(msg2a));
1   CALL print$msg1;
1   DO I = 1 TO LENGTH(msg3);                           /* query for initial time */
2       CALL co(msg3(I-1));
2       END;
1   CALL print$msg1;
1   DO WHILE not$done;                                  /* input initial time values */
2       char = ci;
2       IF ((char> =30H) AND (char< =39H)) THEN
    DO;
3           CALL co(char);
3           DO CASE count1;
4               hours = SHL(hours,4) + (char - 30H);    /* input ASCII and convert to BCD */
4               minutes = SHL(minutes,4) + (char - 30H);
4               seconds = SHL(seconds,4) + (char - 30H);
```

280249-17

Figure 20 (Continued)

```
            END;
          END;
        ELSE IF (char = ':') THEN DO;
            count1 = count1 + 1;
            CALL co(char);
            END;
          ELSE IF (char = 0DH) THEN not$done = FALSE;
            ELSE CALL co(bell);
        END;
    CALL print$msg1;
    hours = (SHR(hours,4) * 10) + (hours AND 0FH);      /* convert BCD to hex */
    minutes = (SHR(minutes,4) * 10) + (minutes AND 0FH);
    seconds = (SHR(seconds,4) * 10) + (seconds AND 0FH);
    CALL print$msg1;
    HSO_CMD = 38H;      /* set-up software-timer1 interrupt and TIMER1 as clock source */
    tock = TIMER1 + 62500;                              /* load initial timer count for
                                                           interrupt */
    HSO_TIME = tock;
    INT_MASK = 20H;                                     /* set mask to select only software timer
                                                           interrupts */
    INT_PENDING = 0;                                    /* clear interrupt pending register */
    ENABLE;                                             /* enable interrupts */
    DO FOREVER;                                         /* start the 'clock' */
        IF tick THEN DO;
          tick = FALSE;
          seconds = seconds + 1;
    $CODE
          IF (seconds=60) THEN DO;
            seconds = 0;
            minutes = minutes + 1;
            IF (minutes=60) THEN DO;
              minutes = 0;
              hours = hours + 1;
              IF (hours=24) THEN hours = 0;
              END;
            END;
          CALL ascii(seconds,.msg2(0));                 /* convert hex times to decimal
                                                           ASCII */
          CALL ascii(minutes,.msg2(3));
          CALL ascii(hours,.msg2(6));
          IF not$first THEN DO;
            DO I = 1 TO 8;                              /* backspace to beginning of line */
              CALL co(BS);
              END;
            END;
          DO I = 1 TO LENGTH(msg2);                     /* print the 'clock' time */
            CALL co(msg2(I));
```

280249-18

Figure 20 (Continued)

```
104     4              END;
             $NOCODE
105     3            not$first = TRUE;
106     3            END;
107     2          END;
108     1      END      clock;

PL/M-96 COMPILER                    iSBE-96 Sample Debug Program
                                ASSEMBLY LISTING OF OBJECT CODE

                                                      ;    STATEMENT              81
   01E7     993C0C            R                       CMPB     SECONDS,#3CH
   01EA     D714                                      BNE      @0019
                                                      ;    STATEMENT              83
   01EC     110C              R                       CLRB     SECONDS
                                                      ;    STATEMENT              84
   01EE     170D              R                       INCB     MINUTES
                                                      ;    STATEMENT              85
   01F0     993C0D            R                       CMPB     MINUTES,#3CH
   01F3     D70B                                      BNE      @0019
                                                      ;    STATEMENT              87
   01F5     110D              R                       CLRB     MINUTES
                                                      ;    STATEMENT              88
   01F7     170E              R                       INCB     HOURS
                                                      ;    STATEMENT              89
   01F9     99180E            R                       CMPB HOURS,#18H
   01FC     D702                                      BNE      @0019
                                                      ; STATEMENT                 90
   01FE     110E              R                       CLRB     HOURS
                                                      ; STATEMENT                 93
   0200                          @0019:
   0200     AC0C1C            R                       LDBZE TMP0,SECONDS
   0203     C81C                                      PUSH     TMP0
   0205     C90000            R                       PUSH     #MSG2
   0208     2E60                                      CALL     ASCII
                                                      ;    STATEMENT              94
   020A     AC0D1C            R                       LDBZE TMP0,MINUTES
   020D     C81C                                      PUSH     TMP0
   020F     C90300            R                       PUSH     #MSG2+3H
   0212     2E56                                      CALL     ASCII
                                                      ;    STATEMENT              95
   0214     AC0E1C            R                       LDBZE TMP0,HOURS
   0217     C81C                                      PUSH     TMP0
   0219     C90600            R                       PUSH     #MSG2+6H
   021C     2E4C                                      CALL     ASCII
```

280249-19

Figure 20 (Continued)

```
                                          ;    STATEMENT                 96
    021E     301211         R             BBC  NOTFIRST,0H,@001C
                                          ;    STATEMENT                 99
    0221     B1010A         R             LDB    I,#1H
    0224                       @001D:
    0224     99080A         R             CMPB  I,#8H
    0227     D909                         BH     @001C
    0229     C90800                       PUSH  #8H
    022C     2E0B                         CALL  CO
                                          ;    STATEMENT                100
    022E     170A           R             INCB    I
    0230     D7F2                         BNE    @001D
                                          ;    STATEMENT                102
    0232                       @001C:
                                          ;    STATEMENT                103
    0232     B1010A         R             LDB    I,#1H
    0235                       @001F:
    0235     AC0A1C         R             LDBZE  TMP0,I
    0238     8908001C                     CMP    TMP0,#8H
    023C     D910                         BH      @0020
    023E     AC0A1C         R             LDBZE  TMP0,I
    0241     AF1D00001C     R             LDBZE  TMP0,MSG2[TMP0]
    0246     C81C                         PUSH   TMP0
    0248     2DEF                         CALL   CO
                                          ;    STATEMENT                104
    024A     170A           R             INCB    I
    024C     D7E7                         BNE    @001F
    024E                       @0020:

 MODULE INFORMATION:
     CODE AREA SIZE                = 0231H   561D
     CONSTANT AREA SIZE            = 0022H    34D
     DATA AREA SIZE                = 0000H     0D
     STATIC REGS AREA SIZE         = 0019H    25D
     OVERLAYABLE REGS AREA SIZE    = 0000H     0D
     MAXIMUM STACK SIZE            = 000AH    10D
     145 LINES READ
 PL/M-96 COMPILATION COMPLETE.     0 WARNINGS,    0 ERRORS
```

280249–20

Figure 20 (Continued)

```
MCS-96 MACRO ASSEMBLER      Sample Debug Program -Interrupt Service Routine
DOS MCS-96 MACRO ASSEMBLER, V1.0
SOURCE FILE: TIMER.A96
OBJECT FILE: TIMER.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: <none>
ERR  LOC    OBJECT           LINE        SOURCE STATEMENT
                               1   $TITLE ('Sample Debug Program -Interrupt Service
                                   Routine')
                               2
                               3
                               4   TIMER    MODULE
                               5
                               6   ;Externals
                               7
                               8   EXTRN         tick       :BYTE ;tick is declared FAST
                                                            so will be in internal RAM
                               9   EXTRN         tock       :WORD ;contains first
                                                            HSO_TIME setting
                              10
                              11   ;Publics
                              12
                              13   PUBLIC        count
                              14
                              15   ;Local variables
                              16
       0004                   17   HSO_TIME      EQU        04H:WORD ; Write only
       0006                   18   HSO_CMD       EQU        06H:BYTE ; Write only
       000A                   19   TIMER1        EQU        0AH:WORD ; Read only
                              20
     0000                     21   RSEG
                              22
     0000                     23   count:        DSB        1
                              24
                              25   ;vector table - only the software timer should be accessed
                              26
     2000                     27   CSEG at 2000h
                              28
     2000   1800          R   29       DCW       oops       ;timer_Overflow
     2002   1800          R   30       DCW       oops       ;ADdone
     2004   1800          R   31       DCW       oops        ;HSI_Data_Available
     2006   1800          R   32       DCW       oops       ;HSO_Execution
     2008   1800          R   33       DCW       oops       ;HSI0
     200A   0000          R   34       DCW       tovfl      ;SW_timers
     200C   1800          R   35       DCW       oops       ;Serial_IO
     200E   1800          R   36       DCW       oops       ;External_Interrupt
                              37
                              38   ;service routines
                                                                          280249-21
```

Figure 21

```
0000                         40  CSEG
                             41
0000   F2                    42  Tovfl:     PUSHF
0001   1700           R      43             INCB    count
0003   990800         R      44             CMPB    count,#8
0006   D706                  45             JNE     loop1
0008   B1FF00         E      46             LDB     tick,#0FFH ;set 'tick' = TRUE
000B   B10000         R      47             LDB     count,#0
000E   B13806                48  loop1:     LDB     HSO_CMD,#38H ;reload HSO
                                                    CAM
0011   4524F40004     E      49             ADD     HSO_TIME,tock,#62500
0016   F3                    50             POPF
0017   F0                    51             RET
                             52
0018   F2                    53  Oops:      PUSHF   ;arriving here means an
                                                    interrupt occurred which
0019   FD                    54             NOP     ; should not have occurred.
                                                    This is also used to
001A   FD                    55             NOP     ; initialize all the interrupt
                                                    vectors for bebugging
001B   F3                    56             POPF    ; with the iSBE-96.
001C   F0                    57             RET
                             58
001D                         59  END

MCS-96 MACRO ASSEMBLER    Sample Debug Program -Interrupt Service Routine

SYMBOL TABLE LISTING
--------------------

            N A M E                          VALUE     ATTRIBUTES
COUNT . . . . . . . . . . . . . . . . . .   0000H     REG   REL   PUBLIC   BYTE
HSO_CMD . . . . . . . . . . . . . . . . .   0006H     NULL  ABS   BYTE
HSO_TIME . . . . . . . . . . . . . . . .    0004H     NULL  ABS   WORD
LOOP1 . . . . . . . . . . . . . . . . . .   000EH     CODE  REL   ENTRY
OOPS . . . . . . . . . . . . . . . . . .    0018H     CODE  REL   ENTRY
TICK . . . . . . . . . . . . . . . . . .    -----     NULL  EXTERNAL  BYTE
TIMER . . . . . . . . . . . . . . . . . .   -----     MODULE  STACKSIZE(0)
TIMER1 . . . . . . . . . . . . . . . . .    000AH     NULL  ABS   WORD
TOCK . . . . . . . . . . . . . . . . . .    -----     NULL  EXTERNAL  WORD
TOVFL . . . . . . . . . . . . . . . . . .   0000H     CODE  REL   ENTRY
ASSEMBLY COMPLETED,        NO ERROR(S) FOUND.
```

280249-22

Figure 21 (Continued)

Since it is much easier to have an integer number for a loop counter, by setting the number of time-outs to eight we find that the count needed is 62,500. This number may eventually have to be tweaked because we did not account for the time required to service the interrupt itself or the tolerance of the 12 MHz crystal on the iSBE-96 emulator, but for our purposes it is close enough.

After prompting for the initial time, the sample program converts the input ASCII characters to hexadecimal. It then initializes software timer 1 to use TIMER 1 as a clock source and signal for an interrupt upon reaching the specified time (a count of 62,500), which is then input to the HSO time register. The software timer interrupt service routine keeps count of the number of times it is activated and on the eighth pass it sets a flag which allows the mainline program to increment the 'clock'. The current 'clock' time is then converted to decimal ASCII and displayed on the terminal connected to the iSBE-96 DTE serial port.

Sample Debug Session

After generating the files CLOCK.P96 and TIMER.A96 as shown in Figures 20 and 21 respectively, use the DOS batch file as show to generate the absolutely located object code (CLOCK.ABS). Figure 20 contains a partial assembly code listing of the PL/M-96 program module (compiled with the CODE and NOCODE controls). The code listing is needed for debugging with the iSBE-96 emulator since it does not support PL/M-96 symbols or line numbers. For the sake of a manageable illustration only part of the assembly code was generated for the PL/M-96 module. The segment map and symbol table generated by RL96 for the sample program (CLOCK.M96) is shown in Figure 22. The segment map shows the address of the instructions of the program since the addresses of the relocatable code in the listing are only relative module addresses.

Once the linked object module has been generated, invoke the iSBE-96 driver software which will sign on with the version number and establish communications with the iSBE-96 emulator. The sample program can then be loaded by typing LOAD CLOCK.ABS <cr>. After the sample program object code has been loaded, begin emulation by typing GO.

You will now be prompted on the terminal to set the current time, 'set time hh:mm:ss <cr>' where <cr> represents a carriage return or enter. After entering the time and carriage return, you will notice that the 'clock' display appears to backup across the screen on the terminal. If you look closely, the hours and seconds also appear to be transposed. Press the escape key on the IBM PC XT or AT, (referred to from now on as the console) to stop emulation. It should be clear that our sample program has two separate problems, relative clock print-out position and transposed hours and seconds.

First let's tackle the print-out position problem. By referring to the PL/M-96 module listing (Figure 20), we discover that the current time is printed out by the DO loop in lines 102 through 104. If you compare these lines with procedure 'print$msg1', you will see that the message index in line 103 should be I-1. This would cause us to only print out 7 of the eight characters. But, the DO loop in lines 98 through 100 backspaces eight characters. These could very well cause the position problem.

To confirm this we first need to consult the assembly code listing section of the PL/M-96 module listing and the link map (Figures 20 and 23), to obtain the address of line 102. The associated line number is printed on the right-hand margin in the assembly code section of the PL/M-96 listings (Figure 20). Since PL/M-96 always places procedures and constants at the beginning of code, the start address for line 102 is 0232H + 2084H = 22B6H. To verify this we can type DASM 22B6 to 22D5 on the console. The resultant dissassembly display is shown in Figure 24. After comparing the display to the listing we can verify that we have the correct address.

To correct the problem we need to load TMP0 (1CH) with I-1 (2EH) and, because TMP0 is then used as an index, we need to ensure that the high byte (1DH) for word pointer 1CH is clear. As you probably already have guessed, the three byte instruction at 22C2H does not give us enough room to do all that. Therefore, we must branch to a non-used area (above 230DH from the link map), add the necessary instructions, and then branch back into the instruction stream. This can be done by typing the following on the console:

```
ASM 22C2 = BR +4AH <cr>,<cr>
ASM 230E = LDB 1C,2E <cr>
DECB 1C <cr>
CLRB 1D <cr>
BR -53H <cr>,<cr>
```

```
plm96 clock.p96
asm96 timer.a96
rl96 clock.obj,timer.obj,plm96.lib to clock.abs stacksize(20H)
```

Figure 22

```
DOS MCS-96 RELOCATOR AND LINKER, V2.0
Copyright 1983 Intel Corporation

INPUT FILES: CLOCK.OBJ, TIMER.OBJ, PLM96.LIB
OUTPUT FILE: CLOCK.ABS
CONTROLS SPECIFIED IN INVOCATION COMMAND:
    STACKSIZE(20H)


INPUT MODULES INCLUDED:
    CLOCK.OBJ(CLOCK)     01/14/86     13:28:27
    TIMER.OBJ(TIMER)     01/14/86     13:28:38
    PLM96.LIB(PLMREG)    11/02/83
    PLM96.LIB(TIME)      11/02/83

SEGMENT MAP FOR CLOCK.ABS(CLOCK):

                        TYPE    BASE     LENGTH   ALIGNMENT   MODULE NAME
                        ----    ----     -------  ----------  --------------

**RESERVED*                     0000H    001AH
                        REG     001AH    0001H    BYTE        TIMER
*** GAP ***                     001BH    0001H
                        REG     001CH    0008H    ABSOLUTE    PLMREG
                        REG     0024H    0019H    WORD        CLOCK
*** GAP ***                     003DH    0001H
                        STACK   003EH    0020H    WORD
*** GAP ***                     005EH    1F86H
                        DATA    1FE4H    0002H    ABSOLUTE    CLOCK
                        DATA    1FE6H    0002H    ABSOLUTE    CLOCK
*** GAP ***                     1FE8H    0018H
                        CODE    2000H    0010H    ABSOLUTE    TIMER
*** GAP ***                     2010H    0070H
                        CODE    2080H    0003H    ABSOLUTE    CLOCK
*** GAP ***                     2083H    0001H
                        CODE    2084H    0253H    WORD        CLOCK
                        CODE    22D7H    001DH    BYTE        TIMER
                        CODE    22F4H    0019H    BYTE        TIME
***GAP ***                      230DH    DCF3H

ATTRIBUTES          VALUE       NAME

SYMBOL TABLE FOR CLOCK.ABS(CLOCK):

                                PUBLICS:
REG       BYTE      0033H          TICK
REG       WORD      002CH          TOCK
CODE      ENTRY     20A6H          CI
```

280249-23

Figure 23

```
CODE       ENTRY      20BDH      CO
CODE       ENTRY      20DAH      INITDT
CODE       ENTRY      20EEH      ASCII
REG        BYTE       001AH      COUNT
REG        NULL       001CH      PLMREG
CODE       ENTRY      22F4H      ??TIME
NULL       NULL       005EH      MEMORY
NULL       NULL       1F86H      ?MEMORY__SIZE

                                MODULE: CLOCK

                                MODULE: TIMER

                                MODULE: PLMREG

                                MODULE: TIME

RL 96 COMPLETED,        0 WARNING(S),       0 ERROR(S)
```

Figure 23 (Continued)

```
*dasm 22b6 to 22d5

ADDRESS      DATA          MNEMONIC      OPERANDS
22B6H        B1012E          LDB         2E,#01
22B9H        AC2E1C          LDBZE       1C,2E
22BCH        8908001C        CMP         1C,#0008
22C0H        D910            JH          $+12
22C2H        AC2E1C          LDBZE       1C,2E
22C5H        AF1D24001C      LDBZE       1C,0024[1C]
22CAH        C81C            PUSH        1C
22CCH        2DEF            SCALL       $-020F
22CEH        172E            INCB        2E
22D0H        D7E7            JNE         $-17
22D2H        B1FF36          LDB         36,#FF

*
```

Figure 24

We must now restart emulation to see if this patch fixes the position problem. To restart emulation type GO FROM 2080 on the console. After setting the time on the terminal, we see that this did fix the position problem.

Now to fix the problem with the hours and seconds transposed on the 'clock' print-out. By consulting the PL/M-96 module listing (Figure 20), we see that the times are converted and put into printable message format by lines 93 through 95. Comparing those lines with the format declarations of messages 2a and 3 in line 2, we see that lines 93 and 95 use the wrong index into message 2 for storing seconds and hours.

To confirm this we again need to consult the assembly code listing section of the PL/M-96 module listing and the link map (Figures 20 and 23), to obtain the address of line 93. The address for line 93 turns out to be 0200H + 2084H = 2284H. We verify this by typing DASM 2284 TO 22AA on the console. After comparing the resultant display (Figure 25) and the code listing, we can see that we have the correct address. To correct the problem we need to swap the instruction at 2289H with the instruction at 229DH. This can be done by typing the following on the console:

```
ASM 2298 = PUSH #2A <cr>,<cr>
ASM 2290 = PUSH #24 <cr>,<cr>
```

We must now restart emulation to see if this fixes the problem. To restart emulation where we left off, type GO on the console. Checking the terminal, we can see that this does fix the transposition problem and the 'clock' print-out is correct.

Now that we have confirmed that our fixes correct the problems, the PL/M-96 module should be updated to incorporate those corrections. The debugged PL/M-96 module is illustrated in Figure 26.

```
*dasm 2284 to 22aa

ADDRESS     DATA       MNEMONIC     OPERANDS
2284H       AC301C       LDBZE      1C,30
2287H       C81C         PUSH       1C
2289H       C92400       PUSH       #0024
228CH       2E60         SCALL      $-019E
228EH       AC311C       LDBZE      1C,31
2291H       C81C         PUSH       1C
2293H       C92700       PUSH       #0027
2296H       2E56         SCALL      $-01A8
2298H       AC321C       LDBZE      1C,32
229BH       C81C         PUSH       1C
229DH       C92A00       PUSH       #002A
22A0H       2E4C         SCALL      $-01B2
22A2H       303611       JBC        36,00,$+14
22A5H       B1012E       LDB        2E,#01
22A8H       99082E       CMPB       2E,#08

*
```

Figure 25

```
$title (' iSBE-96 Sample Debug Program')
$optimize (3)
clock: DO;
/* local declarations */
DECLARE   bell                  LITERALLY   '07H',
          BS                    LITERALLY   '08H',
          FOREVER               LITERALLY   'WHILE 1',
          FALSE                 LITERALLY   '0',
          TRUE                  LITERALLY   'NOT FALSE',
          BOOLEAN               LITERALLY   'BYTE',
          msg1(*)               BYTE        DATA(0dH,0aH),
          msg2a(*)              BYTE        DATA(0,0,':',0,0,':',0,0),
          msg2(8)               BYTE        FAST,
          msg3(*)               BYTE        DATA('set time - hh:mm:ss <cr>'),
          (I,char)              BYTE,
          seconds               BYTE,
          minutes               BYTE,
          hours                 BYTE,
          tick                  BYTE        FAST       PUBLIC,
          tockWORDPUBLIC,
          count                 BYTE        EXTERNAL,
          count1                BYTE,
          not$done              BOOLEAN,
          not$first             BOOLEAN,
          HSO_TIME              WORD        AT (04H),
          HSO_CMD               BYTE        AT (06H),
          INT_MASK              BYTE        AT (08H),
          INT_PENDINGBYTEAT (09H),
          TIMER1                WORD        AT (0AH),
          dt_data               ADDRESS     AT (1FE4H),
          dt_status             ADDRESS     AT (1FE6H);
/* Procedure declarations */
ci:     PROCEDURE     BYTE      PUBLIC;
DO WHILE ((dt_status AND 02H) = 0H);                /* wait till RxRDY */
    END;
char = dt_data AND 7FH;
RETURN char;
END   ci;
co:     PROCEDURE       (char)      PUBLIC;
DECLARE     char     BYTE;
DO WHILE ((dt_status AND 1) = 0 );                  /* wait till TxRDY */
    END;
dt_data = char;
END     co;
init$DT:     PROCEDURE        PUBLIC;
dt_status = 37H;                                    /* clear any errors on the DTs 8251A USART */
```

280249–24

Figure 26

```
CALL TIME(1);
char = dt_data;                                         /* clear the DT receive buffer */
END     init$DT;

ascii:    PROCEDURE (value,dest$ptr)      PUBLIC;
DECLARE (value,temp)       BYTE,
          dest$ptr        ADDRESS,
          (dest     BASED    dest$ptr)  (2)    BYTE;
value = SHL((value/10),4) + (value MOD 10);             /* convert to BCD */
temp = value;
dest(0) = SHR(temp,4) + 30H;                            /* convert to ASCII decimal value */
dest(1) = (value AND 0FH) + 30H;
END ascii;

print$msg1:    PROCEDURE;
DECLARE    I    BYTE;
DO I = 1 TO LENGTH(msg1);
    CALL co(msg1(I-1));
    END;
END     print$msg1;

/* Program starts here */
CALL TIME(50);                                          /* delay to insure iSBE-96 NMI line is stable */
CALL init$DT;                                           /* initialize DT serial port */
count,count1 = 0;                                       /* initialize variables */
not$done = TRUE;
not$first,tick = FALSE;
seconds,minutes,hours = 0;
CALL movb(.msg2a,.msg2,LENGTH(msg2a));
CALL print$msg1;
DO I = 1 TO LENGTH(msg3);                               /* query for initial time */
    CALL co(msg3(I-1));
    END;
CALL print$msg1;
DO WHILE not$done;                                      /* input initial time values */
    char = ci;
    IF ((char> =30H) AND (char< =39H)) THEN DO;
      CALL co(char);
      DO CASE count1;
        hours = SHL(hours,4) + (char - 30H);            /* input ASCII and convert to BCD */
        minutes = SHL(minutes,4) + (char - 30H);
        seconds = SHL(seconds,4) + (char - 30H);
        END;
      END;
    ELSE IF (char = ':') THEN DO;
        count1 = count1 + 1;
        CALL co(char);
        END;
      ELSE IF (char = 0DH) THEN not$done = FALSE;
```

280249-25

Figure 26 (Continued)

```
            ELSE CALL co(bell);
        END;
    CALL print$msg1;
    hours = (SHR(hours,4) * 10) + (hours AND 0FH);          /* convert BCD to hex */
    minutes = (SHR(minutes,4) * 10) + (minutes AND 0FH);
    seconds = (SHR(seconds,4) * 10) + (seconds AND 0FH);
    CALL print$msg1;
    HSO_CMD = 38H;                                          /* set-up software-timer1 interrupt and
                                                            TIMER1 as clock source */
    tock = TIMER1 + 62500;                                  /* load initial timer count for interrupt */
    HSO_TIME = tock;
    INT_MASK = 20H;                                         /* set mask to select only software timer
                                                            interrupts */
    INT_PENDING = 0;                                        /* clear interrupt pending register */
    ENABLE;                                                 /* enable interrupts */
    DO FOREVER;                                             /* start the 'clock' */
        IF tick THEN DO;
          tick = FALSE;
          seconds = seconds + 1;
    $CODE
          IF (seconds = 60) THEN DO;
            seconds = 0;
            minutes = minutes + 1;
            IF (minutes = 60) THEN DO;
              minutes = 0;
              hours = hours + 1;
              IF (hours = 24) THEN hours = 0;
              END;
            END;
    CALL ascii(seconds,.msg2(6));                           /* convert hex times to decimal ASCII */
    CALL ascii(minutes,.msg2(3));
    CALL ascii(hours,.msg2(0));
          IF not$first THEN DO;
            DO I = 1 TO 8;                                  /* backspace to beginning of line */
              CALL co(BS);
              END;
            END;
          DO I = 1 TO LENGTH(msg2);                         /* print the 'clock' time */
            CALL co(msg2(I-1));
            END;
    $NOCODE
          not$first = TRUE;
          END;
        END;
    END clock;
```

280249–26

Figure 26 (Continued)

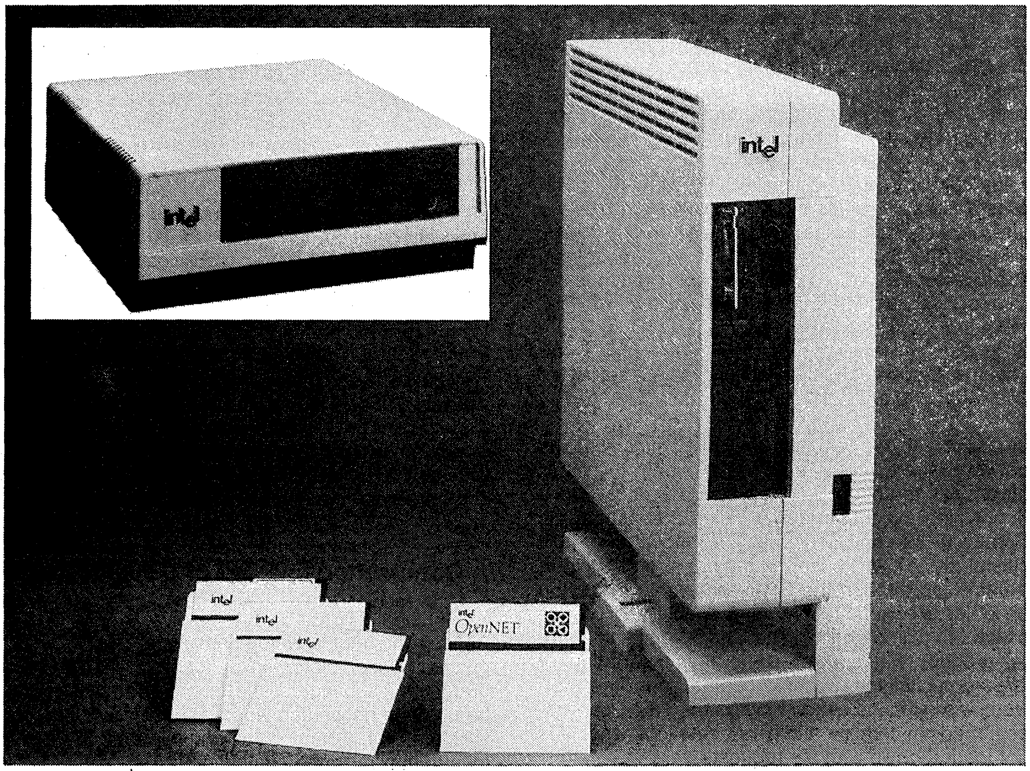
Network Development Systems

4

PRELIMINARY

OpenNET™ NETWORK RESOURCE MANAGER (NRM) iMDX 460

- Ethernet-Based File Server to Connect
  - — Intellec Series II/III/IV
  - — Model 800
  - — Compil$_{e}$ngine
  - — Intel XENIX* and iRMX™-Based System
  - — VAX*/MicroVAX* Running VMS
  - — DOS 3.1 (and up)-Based PCs
- Runs 8- and 16-Bit Languages
- Protected Hierarchical File System
- Remote Job Execution for Distributed Processing
- Shared Resources
  - — Central Disk Storage up to 560 MB
  - — 60 MB Streaming Tape
  - — Shared Spooled Line Printer
- High-Performance Hardware
  - — iSBC® 286/12, 8 MHz CPU Board with One Megabyte of Zero Wait-State RAM
  - — iSBC® 214 Disk Controller with a Dedicated 80186 CPU and Track Caching



280173–1

*XENIX is a registered trademark of the Microsoft Corp.
*VAX, MicroVAX, VMS and DEC are registered trademarks of Digital Equipment Corp.

November 1986
Order Number: 280173-002

OVERVIEW

The OpenNET NRM is a network file server optimized for a distributed development systems environment. Workstations on the network address the hardware engineer's needs by providing the base environment for tools such as in-circuit emulators and PROM programmers. Furthermore, workstations support software engineering teams—hosting software tools such as compilers, software debuggers, and project management tools. For a project team, the OpenNET NRM creates a network that provides the dual advantage of a powerful desk-top computer and access to shared resources on a high-speed standard network. These capabilities include transparent file access to shared files, remote job execution, and print spooling. The OpenNET NRM is plug-compatible with the original NDS-II NRM.

FUNCTIONAL DESCRIPTION

The OpenNET NRM manages all workstation requests for centralized network resources. These tasks include service of workstation file access requests, print spooling, management of remote job execution queues, and network maintenance functions such as user creation/authentication, file archiving, and system configuration.

The iNDX operating system on the OpenNET NRM uses a hierarchical file system providing file sharing and protection features. With this file organization, files can be logically grouped into directories and sub-directories. File protection is implemented in the form of access rights for each file or directory.

For messages or simple file transfers among workstations, Electronic Mail is included for Series II/III/IV, Model 800 and ISIS cluster users.

In addition to the management of communications between shared disks and workstations, the NRM maximizes the use of all network resources with a remote job execution facility called Distributed Job Control (DJC). Any Series II/III/IV, Model 800, ISIS cluster or VAX/VMS user on the network can export a batch job through the NRM for remote execution on other idle workstations or a specialized system called the Compil$_e$ngine. The Compil$_e$ngine is an Intel product that specializes in the importation of large compilations and link/locates from other network workstations. Alternatively, the NRM can accept medium compile and link/locate jobs when it is not loaded down with file requests.

NDS-II Plus OpenNET™ Network Communications Enable Data Sharing in Mixed Vendor Environments

The OpenNET NRM communicates with other systems over the industry standard Ethernet network. The OpenNET NRM supports two Intel network protocols used on Ehternet: the NDS-II and OpenNET network protocols. Installed in the NRM, and iSBC 550 communications board set supports the NDS-II protocols. An iSXM 552 supports the ISO 8073 compatible iNA960 transport software for the OpenNET network. Intellec Series II/III/IV systems, Model 800 workstations, Compil$_e$ngines, communicate with the NRM via the NDS-II protocols while XENIX, iRMX and PC-DOS systems communicate via the OpenNET network. The VAX and MicroVAX running VMS connect to the NRM via VAX Link.

All data that is stored at the NRM is visible to, and can be transparently accessed by, all workstations in the network. VAX/VMS users have file transfer capability. For example, PCs or 286/310 systems (running the iRMX or XENIX operating system) can share files on the NRM with Series IVs with the capability to upload those files to the VAX for back-up.

Network Managment Facilities

The NRM provides your network administrator with tools such as hardware diagnostics, network status and configuration commands to help optimize your user environment. These capabilities include the display all network users and transaction counts, definition of the number of multitasking jobs and allocate communication memory. For example, for a network with 2–3 Series IVs and 4–5 PCs, the NRM can be configured with maximum communication buffer memory. This is to maximize throughput for a few workstations. Alternatively, for a network with 4–5 Series IVs and 20 PCs, the NRM can be configured with smaller buffers but large numbers of users. This is to maximize the number of users on the network.

A Comprehensive Set of Development Tools Supported

All current Intel development tools such as languages, assemblers, linker/locators, PROM programmers, debuggers and in-circuit emulators can be used on networked workstations. In addition, the NRM can run 8- and 16-bit compile and link/locate jobs to maximize the computing power of your NRM when it is relatively idle from communication tasks. These jobs can be sent from any workstation on the network.

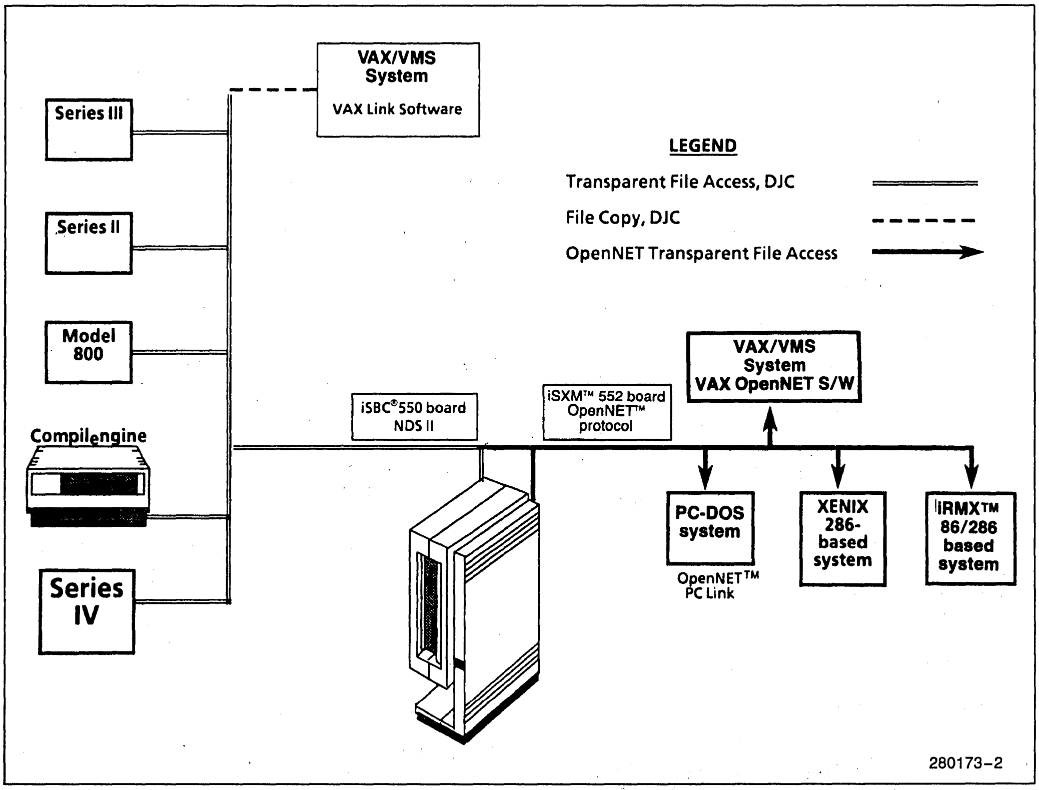


Figure 1. Share Files Transparently on the OpenNET™ NRM From Any Workstation

State-of-the-Art System Components means Performance

The OpenNET NRM is a high performance 80286-based microcomputer. Two models are offered: the MAXI and MINI. The MAXI has a fast 140 MB disk with a 60 MB streaming tape drive while the MINI has a 40 MB disk with no tape. Each system contains an iSBC 286/12 8 MHz CPU board with one megabyte of zero wait-state on-board Random Access Memory (RAM). Also contributing to the high system performance is the iSBC 214 multiprocessing peripheral controller. This controller features its own 80186 microprocessor and 32 KB of software transparent cache memory. The 80186 offloads the 80286 CPU from virtually all peripheral controller tasks, while the cache memory greatly reduces apparent access times to hard disk memory. The iSBC 214 controller is used to support the Winchester and 640 KB half-height floppy disk drives in both the MAXI and MINI models. The iSBC 214 controller additionally supports the streaming tape drive in the MAXI model.

Interconnecting Hardware Is Standard

To connect workstations to the OpenNET NRM, standard Ethernet transceivers, transceiver cables, and Ethernet coaxial cables are used. Intel's Intellink module may also be used to connect multiple workstations to Ethernet. The same (one) Intellink module can be used to connect Series II/III/IVs, VAXs via VAX Link, and OpenNET workstations, as long as cables from both the iSBC 550 and iSXM 522 boards are connected to that Intellink module. The Series II/III/IVs, VAXs will respond to messages from the iSBC 550 board while the OpenNET workstations will respond to messages from the iSXM 552 board. This helps the user optimize usage of current and new interconnect hardware.

The OpenNET™ NRM Expands To Fit Growing Development Environments

Creating a network in your development lab is accomplished by adding an NRM and upgrade kits for Intellec, 286/310 XENIX and iRMX, PC-DOS and VAX/VMS systems. The network grows with your development environment. Workstations can be added, mass storage increased, and multiple networks connected—giving your development environment maximum flexibility. You only need to acquire as much development equipment as you require today, knowing that you will be able to grow in increments tomorrow.

SPECIFICATIONS

Hardware

Dimensions:	6½" x 17" x 22"
Weight:	55 lbs.
Electrical:	User selectable AC power with either 88-132V, 60 Hz or 180-264V, 50 Hz

Software

iNDX R3.2 Operating System

OPERATING ENVIRONMENT

Environmental Characteristics

Temperature:	10°C to 35°C
Humidity:	20-80% relative humidity
Altitude:	Sea Level to 8000 Feet

Hardware Required

User supplied ANSI 3.64 standard terminal
Workstation interconnecting hardware:
Intellink module, transceivers, transceiver cables, Ethernet coaxial cables (depending on number of workstations and distance)

NDS-II Workstations

Workstation	Hardware/Software Products Required
Compil$_e$ngine	—
Model 800	PIMDX455
Series II/III	PIMDX455
Series IV	PIMDX456
VAX/VMS	IMDX 392 & DEUNA* board
μVAX/VMX	iMDX 392 & DEQNA* board

*A DEC product

OpenNET™ Workstations

Workstation	Hardware/Software Products Required
PC-DOS V3.1 System	PCLNK
System 310 XENIX	XNX-NET & iSXM 552 board
System 310 iRMX 86/286	iRMX-NET 8 iSXM 552 board
VAX/VMS	VMSSVR
μVAX/VMS	MVSSVR

DOCUMENTATION

iNDX User's Guide
(order #: 138809-001)

iNDX System Installation Guide
(order #: 138810-001)

ORDERING INFORMATION

Product Order Code	Description
iMDX 460-140T	OpenNET NRM (MAXI model)
iMDX 460-40	OpenNET NRM (MINI model)
iSYP 312	Floor stand which encloses either the OpenNET NRM or the SYS 311 (see peripheral upgrades section) peripheral expansion box

Interconnecting Hardware

Product Order Code	Description
PIMDX 457/458	Transceiver cables (10/50 meters) (two are required for an OpenNET NRM)
PMDX 3015	Transceiver for Ethernet coaxial cables (at least two are required unless an Intellink is used)
iDCM 91-1	Intellink module (the OpenNET NRM uses two ports)
PIMDX 3016-1/ 3016-2	Ethernet coaxial cable (25/50 meters)

Workstation Kits

Product Order Code	Description
PIMDX 455	NDS-II Workstation Upgrade Kit for any Series II/85, Series III, or Model 800 to connect to the OpenNET NRM
PIMDX 456	NDS-II Workstation Upgrade Kit for the Series IV
PIMDX 581	ISIS Cluster Board Package
MVMSSVR	VAX/VMS-OpenNET Link S/W and Controller Board
VMSSVR	MicroVAX/VMS OpenNET Link S/W and Controller Board
PCLNK	OpenNET PC Link hardware and software kit to connect the PC XT, PC AT, and compatible systems to the NRM via the OpenNET network; requires DOS 3.1 or higher
RMXNT961KITWSU	iRMX Networking Software for a 286/310 system running the iRMX 86 Operating System to connect to the NRM via the OpenNET network
SXM 5524	Ethernet-based Single Board Transport Engine for 310 systems
XNXNETKRIKIT	OpenNET-XenixNET iNA 961 iSXM 552 and XenixNET Pass-through Kit

STORAGE EXPANSION SUB-SYSTEMS

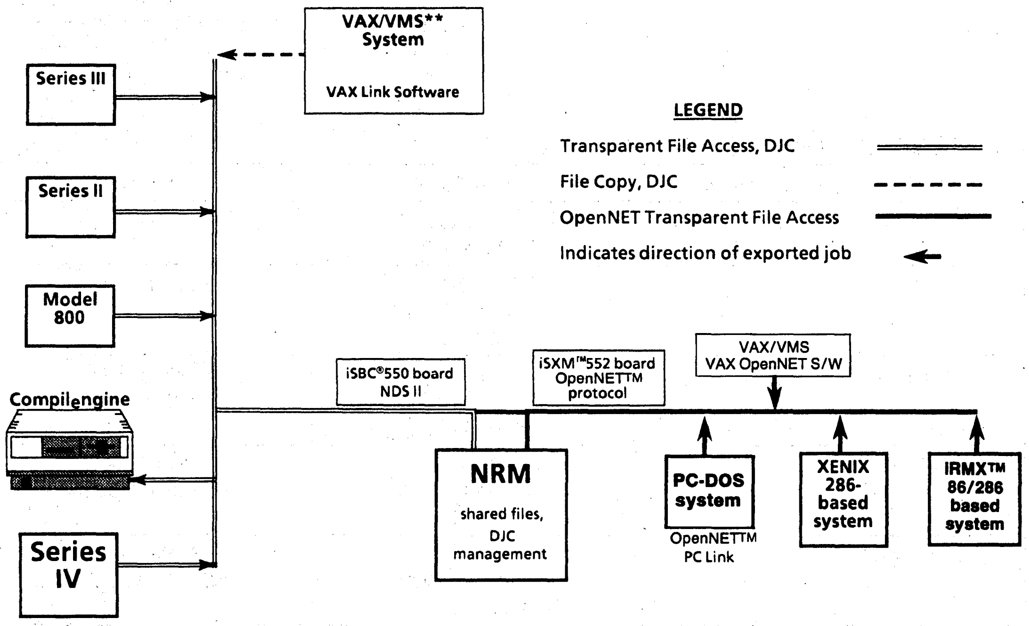
Winchester Storage Size	With Tape	No Tape
0 MB	PSYS311A02 PSXM311TCBL	—
40 MB	PSYS311A14 PSXM311WDCBL PSXM311TCBL	PSYS311A13 PSXM311WDCBL
2 x 40 MB	PSYS311A17 PSXM311WDCBL PSXM311TCBL	PSYS311A16 PSXM311WDCBL
140 MB	PSYS311A34 PSXM311WDCBL PSXM311TCBL	PSYS311A33 PSXM311WDCBL
2 x 140 MB	PSYS311A37 PSXM311WDCBL PSXM311TCBL	PSYS311A36 PSXM311WDCBL
3 x 140 MB	—	PSYS311A39 PSXM311WDCBL

NOTE:
Check product catalog (under PSYS311) for add-on Winchester disk drive ordering information

COMPILeNGINE iMDX 485CE

- **Fast, Dedicated Import Station on the Network**
- **Off-Loads Compilations and Link/Locates from other Intellec® Development Workstations on the Network Using Remote Job Execution**
- **Off-Loads Compile Jobs from a DEC VAX/VMS** via the NRM and VAX Link**
- **Supports 8051, 8086, 8096, 80186, 80286 Languages Including ASM, C, PL/M, Pascal, and Fortran**
- **High Performance System Containing an 8 MHz 80286 CPU with One Megabyte of Zero Wait-State RAM**
- **Fast Disk Access via iSBC® 214 Disk Controller with 32 KB Track Caching**
- **640 KB Half-Height Floppy Disk Drive for Initial Software Load**
- **Supports Optional Standard Terminal for System Maintenance and Direct Job Execution**

The Compilengine is a 286-based supermicrocomputer system designed to improve productivity of a networked development team. Connected to a Network Resource Manager (NRM), it is optimized to off-load large compile and link/locate jobs from the Series II/III/IV, Model 800, and DEC's VAX/VMS systems. PC-DOS and XENIX* systems connected via the OpenNET™ network to the NRM can also export compiles and link/locates onto the Compilengine. The Compilengine performs compilations and link/locate jobs faster than any single workstation on the network. By exporting time-consuming jobs to a shared Compilengine, workstations are free to perform other tasks such as editing or debugging.

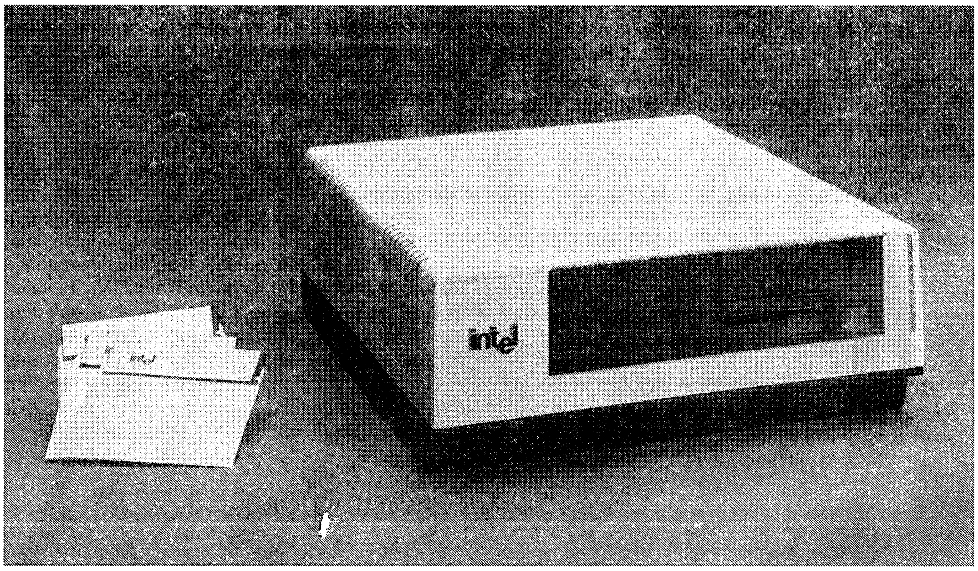


280174–1

Offload Your Non-Interactive Software Jobs from any System to the *Compilengine*

*XENIX is a registered trademark of the Microsoft Corporation.
**VAX, VMS, DEC are registered trademarks of Digital Equipment Corporation.

October 1986
Order Number: 280174-002


280174-2

FUNCTIONAL DESCRIPTION

The Compil$_e$ngine adds more performance to your networked development environment by off-loading time consuming tasks from workstations. These tasks are execued by a powerful 286-based system hardware running the iNDX operating system. By using Intel's remote job execution facility called Distributed Job Control (DJC), Intellec Series II/III/IV and Model 800 workstations as well as VAX/VMS systems connected via optional VAX Link software can export compile and/or link/locate jobs to one or more Compil$_e$ngines. Remote job execution is also possible from OpenNET systems via an NRM supporting the OpenNET network and an export utility in the Network Toolbox product. The user can now compile or link, during the workday, those long, CPU-intensive jobs that traditionally have been executed off-hours.

Easy to Start, Easy to Use

The Compil$_e$ngine is connected to the network just like any other workstation on the NDS-II network, via the iSBC 550 communication board set (included) and a transceiver cable. Once physically connected and configured (system generated) onto the NRM, the Compil$_e$ngine starts automatically with a simple switch on of the power. The system autoboots from the floppy disk drive or the network. Sending jobs to this execution vehicle is equally as simple. By executing the EXPORT command on a batch job containing the compile or link/locate task, DJC on the NRM takes over and completes the job at the Compil$_e$ngine.

The Right Location of Files Maximizes Performance

To maximize the performance of compiles and link/locates sent to the Compil$_e$ngine, frequently accessed but stable files (compiler, linker/locators, and language libraries) should reside on the Compil$_e$ngine's local 40 MB Winchester drive. This will help reduce network traffic. On the other hand, frequently changed files such as source code, include files, and user object libraries should reside on the NRM for version control. For the best performance on large compiles, files may be copied to the Compil$_e$ngine as part of an exported job. All shared files should reside on the NRM so that the most up-to-date copy of the files are visible to all network users. During off-hours the NRM can update or replace compilers, linker/locators, or other commonly used files with the latest versions.

State-Of-The-Art System Components Means Performance

The Compil$_e$ngine is a high-performance super-microcomputer with state-of-the-art technology. Each system contains the advanced iSBC 286/12 8 MHz CPU board with one megabyte of zero wait-state on board Random Access Memory (RAM). Also contributing to the high system performance is the iSBC 214 Multiprocessing Peripheral Controller. This controller features its own 80186 microprocessor and 32 KB of software transparent cache memory. The 80186 offloads the 80286 CPU from virtually all peripheral controller tasks, while the cache memory greatly reduces apparent access times to hard disk memory. The iSBC 214 Controller is used to support a 40 MB Winchester drive and 640 KB 5¼" floppy disk drive. To communicate with the NRM, the Compil$_e$ngine uses the iSBC 550 Communication Board set for the standard high-speed (10 MB per second) Ethernet network.

Comprehensive Software Development Tools Supported

Since the Compil$_e$ngine is a very fast, specialized iNDX system, all languages, macro assemblers, and linker/locaters currently supported on the Series IV and the NRM are also supported on the Compil$_e$ngine. This includes popular high-level languages such as PL/M, Pascal, Fortran, and C, as well as powerful "high-level" macro assemblers such as ASM86. These languages support development for 8051, 8086/8088, 80186/188, 80286, and 8096 architectures.

For More Flexibility

Although a terminal is not required to operate the Compil$_e$ngine, the capability to connect an ANSI standard terminal is provided. This feature gives the customer the ability to perform file maintenance on the local storage devices (40 MB Winchester and floppy disk drives). The user can also initiate jobs directly on the Compil$_e$ngine.

APPLICATIONS

As shown in Figure 1, any workstation configured on the NDS-II network can export jobs to the Compil$_e$ngine using DJC. For example, on a Series IV, the user can simply execute the EXPORT command on a batch file containing a compile job. Then, edit or debug other programs while the Compil$_e$ngine compiles that job.

Use of the Compil$_e$ngine(s) from a VAX connected via VAX Link R2.0 is similar to use from a Series IV. The only requirement is that the source files to be compiled or the object modules/library routines to be link/located reside on the NRM. This is easily achieved by creating a batch file on the VAX. Each time a DJC command for remote compiles is executed under VMS, this batch file copies files to be compiled from the VAX onto the NRM and returns the compiled code back to the VAX. (See Figure 2.)

Systems connected on the OpenNET network can access the Compil$_e$ngine via the NRM. To use the Compil$_e$ngine, the PC or XENIX user simply copies the submit file to a special directory on the NRM. An export program in the Network Toolbox that runs on the NRM will scan that directory and send that job to the DJC queue for remote execution at the Compil$_e$ngine. (See Figure 3.)

SPECIFICATIONS

Hardware

Dimensions:	6½" x 17" x 22"
Weight:	Less than 55 pounds
Electrical:	User selectable AC power with either 88-132V, 60 Hz or 180-264V, 50 Hz

Software

iNDX operating system

OPERATING ENVIRONMENT

Environmental Characteristics

Temperature:	10°C to 35°C
Humidity:	20–80% relative humidity
Altitude:	Sea level to 8000 feet

Hardware Required

- An NDS-II NRM (iMDX 450) or OpenNET NRM (iMDX 460) with the iSBC 550 communications board set installed.
- Interconnecting hardware (one of the following):
  - One transceiver cable and one port on an Intellink module
  - One transceiver cable, one transceiver and an Ethernet coaxial cable
- Optional: ANSI 3.64 standard terminal

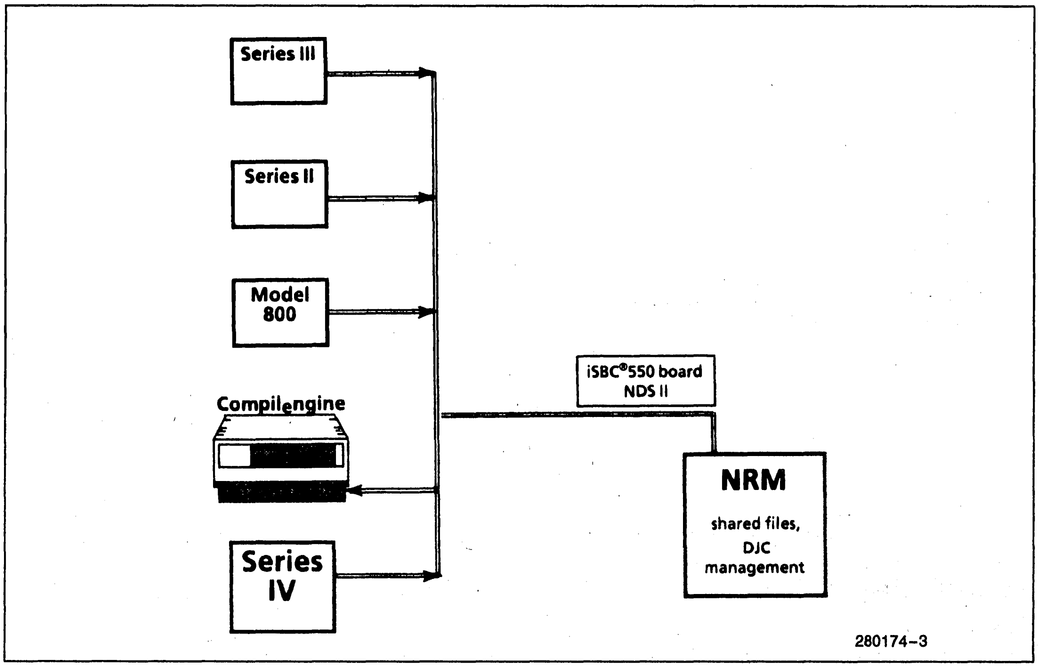


Figure 1. Series II/III/IV and Model 800 Environment

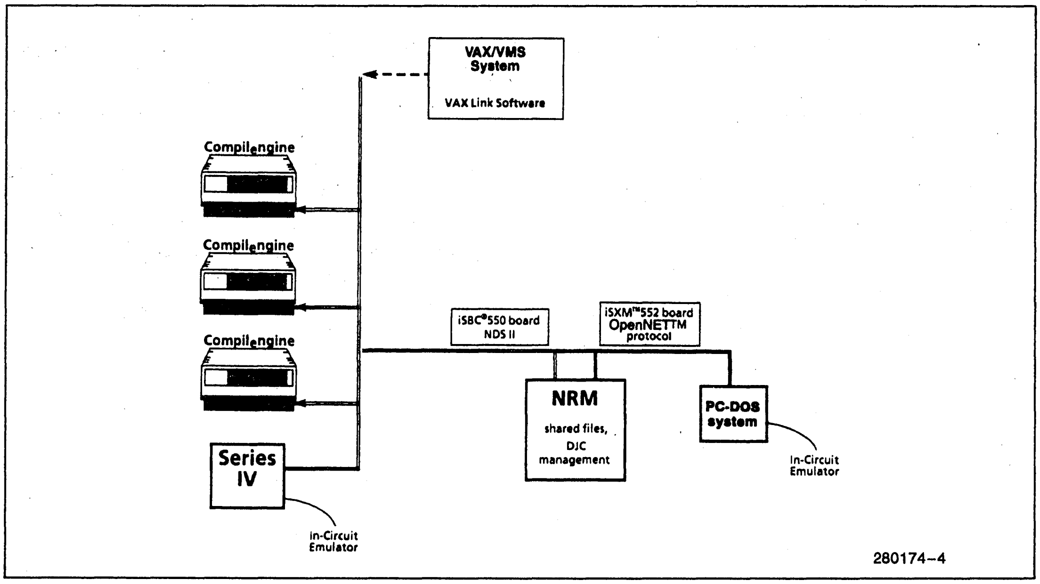


Figure 2. VAX/VMS and Other Intellec® Workstations

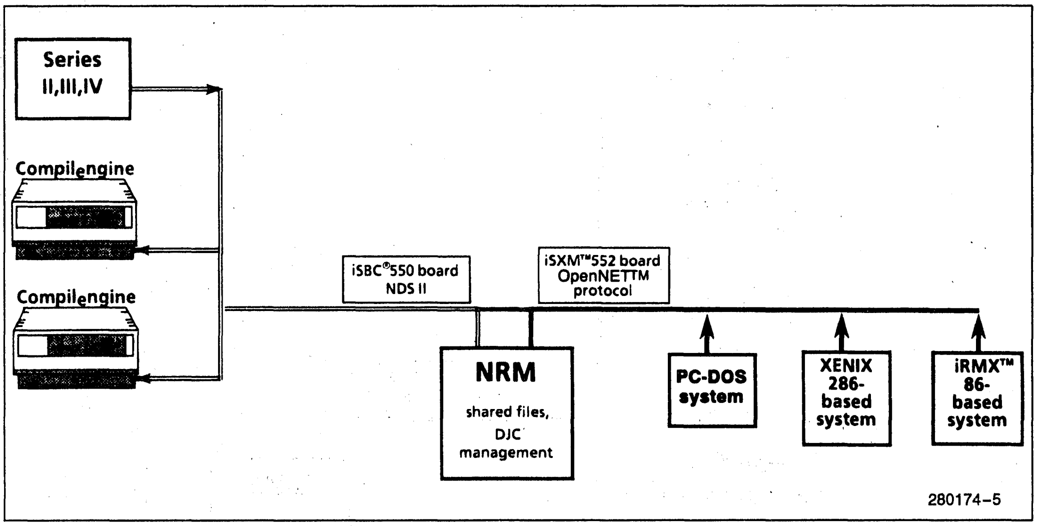


Figure 3. OpenNET™ Systems and Other Intellec® Development Systems

Software Required

High-level language compilers and/or assemblers on 5¼″ iNDX-formatted (96 tpi) diskettes.

SUPPORT DOCUMENTATION

iNDX User's Guide
(order # 138809)

iNDX System Installation Guide
(order # 138810)

ORDERING INFORMATION

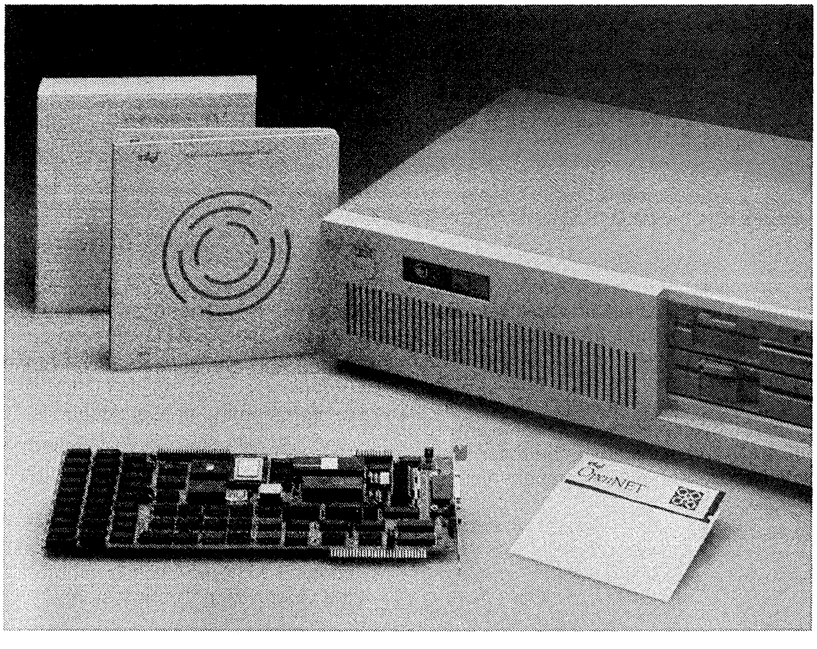
Product Order Code	Description
iMDX 485CE	Compil$_e$ngine
iMDX 460-140T	OpenNET NRM (maxi model)
iMDX 460-40	OpenNET NRM (mini model)
iMDX 457/458	Transceiver cables (10/50 meters)
iDCM 911-1	Intellink module
iMDX 3015	Transceiver for Ethernet coaxial cables
iMDX 3016	Ethernet coaxial cables (25 or 50 meters)
iSYP 312	Floor stand for the Compil$_e$ngine
iMDX460-140T	OpenNET NRM file server (Maxi Model)
iMDX460-40	OpenNET NRM file server (Mini Model)

OpenNET™ PERSONAL COMPUTER LINK

- **Connects an IBM* PC AT, PC XT (and PC-DOS Compatibles) to the OpenNET™ Network**
- **Works with Standard DOS Commands**
- **Interconnects a PC System to iRMX™, XENIX*, and NDS II/NRM Systems Offering OpenNET Server Capability**
- **Uses an 80186/82586 Processor-Based Network Controller Board**
- **Contains Power-Up Diagnostics**
- **Supports the ISO/OSI Seven Layer Networking Standards**
- **Enables a PC System to Access Remote Storage and Printer Devices**
- **Provides Transparent-File-Access Capability Between a PC System and Remote Servers**
- **Uses ISO 8073 Transport and Ethernet/IEEE 802.3 Standard Communication Protocols**
- **Intelligent Board Uses Only 44K of PC's Memory**

The OpenNET Personal Computer Link (OpenNET PC Link) enables users to connect their IBM PC AT and PC XT computer systems to the OpenNET network. This connection enables a PC system to be configured as a consumer workstation on the OpenNET network, and to transparently access and share files and printers on an OpenNET network resource manager (NRM), NDS-II (with the OpenNET upgrade installed) NRM, iRMX, and XENIX-based remote server systems. The OpenNET PC Link is an 80186/82586 microprocessor-based expansion board, which is easily installed in an expansion card slot of the PC system. On-board jumpers and a user configurable software package enable the OpenNET PC Link to be used with a wide-range of expansion boards currently available for the PC system. The OpenNET PC Link incorporates the Microsoft* Networks (MS-NET) networking software and iNA 960 (ISO 8073 compatible) transport software as a part of its software package.

*IBM is a registered trademark of the International Business Machines Corporation.
*MS-DOS is a trademark of the Microsoft Corporation.
*Microsoft is a registered trademark of the Microsoft Corporation.
*XENIX is a trademark of the Microsoft Corporation.



280164–1

October 1986
Order Number: 280164-002

PRODUCT OVERVIEW

The OpenNET PC Link is a member of Intel's OpenNET networking product family. The OpenNET products incorporate a set of system and component level LAN products covering all seven layers of the ISO (International Standards Organization) Open System Interconnect (OSI) model, and the protocols on which they are based. OpenNET network protocols are established industry standards for each function. Therefore, OpenNET network products can connect and operate not only with each other, but with the most popular networking products from other vendors. OpenNET networks provide a high level of interoperability between heterogeneous systems (MS-DOS*, PC-DOS, iNDX, XENIX, and iRMX operating system versions are available). Thus, users can tailor their networks to meet their specific needs by incorporating any combination of the capabilities of these diverse systems.

The OpenNET network application protocols implemented by OpenNET PC Link software are those adopted by Intel, Microsoft, and IBM for their computer networking products. The OpenNET PC Link software is compatible with and will operate with iNDX, XENIX, and iRMX networking software at the application layer.

PHYSICAL DESCRIPTION

The OpenNET PC Link consists of a network controller board and a 5¼ inch disk that contains the software necessary for the PC system to communicate across the OpenNET network. The following sections describe the hardware and software components of the OpenNET PC Link.

OpenNET PC Link Network Controller Board

The network controller board is an adaptor board that can be installed in any available expansion slot of a PC system. The board implements the industry standard ISO 8073 transport protocol (a modified version of iNA 960) and Ethernet/IEEE 802.3 physical data link technology (see Figure 1). The board uses an Intel 80186 microprocessor in combination with an 82586 LAN communication controller. The board includes the following major components:

- 80186 microprocessor
- 82586 LAN communications controller
- 8 KB of EPROM
- 128 KB of RAM shared between the PC system and the 80186 microprocessor on the network controller board
- 82501 Ethernet serial interface
- Fujitsu MB502A encoder decoder
- 15-pin Ethernet D connector
- 8-bit parallel DMA interface and control register set
- Power-up diagnostics

The network controller board performs all network communication functions for the first two layers of the ISO/OSI model (see Figure 2). Layers three and four reside in the modified iNA 960 transport software. The remaining layers (five through seven) reside in the MS-NET networking software on the PC system.

POWER-UP DIAGNOSTICS

An effective diagnostic function is implemented in firmware on the network controller board. This function is invoked at system initializatiion during both power-up and system reset time. The following list summarizes the functions tested:

- 80186 and 82586 microprocessors
- I/O ports
- Shared memory window
- Interrupt channels
- DMA channel
- Ethernet connection

An on-board LED indicates whether the network controller board failed any of the various test functions.

OpenNET PC Link Software

The software is supplied on a 5¼ inch double-density disk (360 KB). The following files are included as part of the OpenNET PC Link:

- A specially configured version of iNA 960 transport layer software, called UBCODE.MEM, which operates on the network controller board.
- A DOS interface driver, called XPORT.EXE, which enables DOS programs to access the network controller board.
- The Microsoft Networks (MS-NET) networking software, release 1.0, which enables users to connect with and access remote file servers on the OpenNET network system.

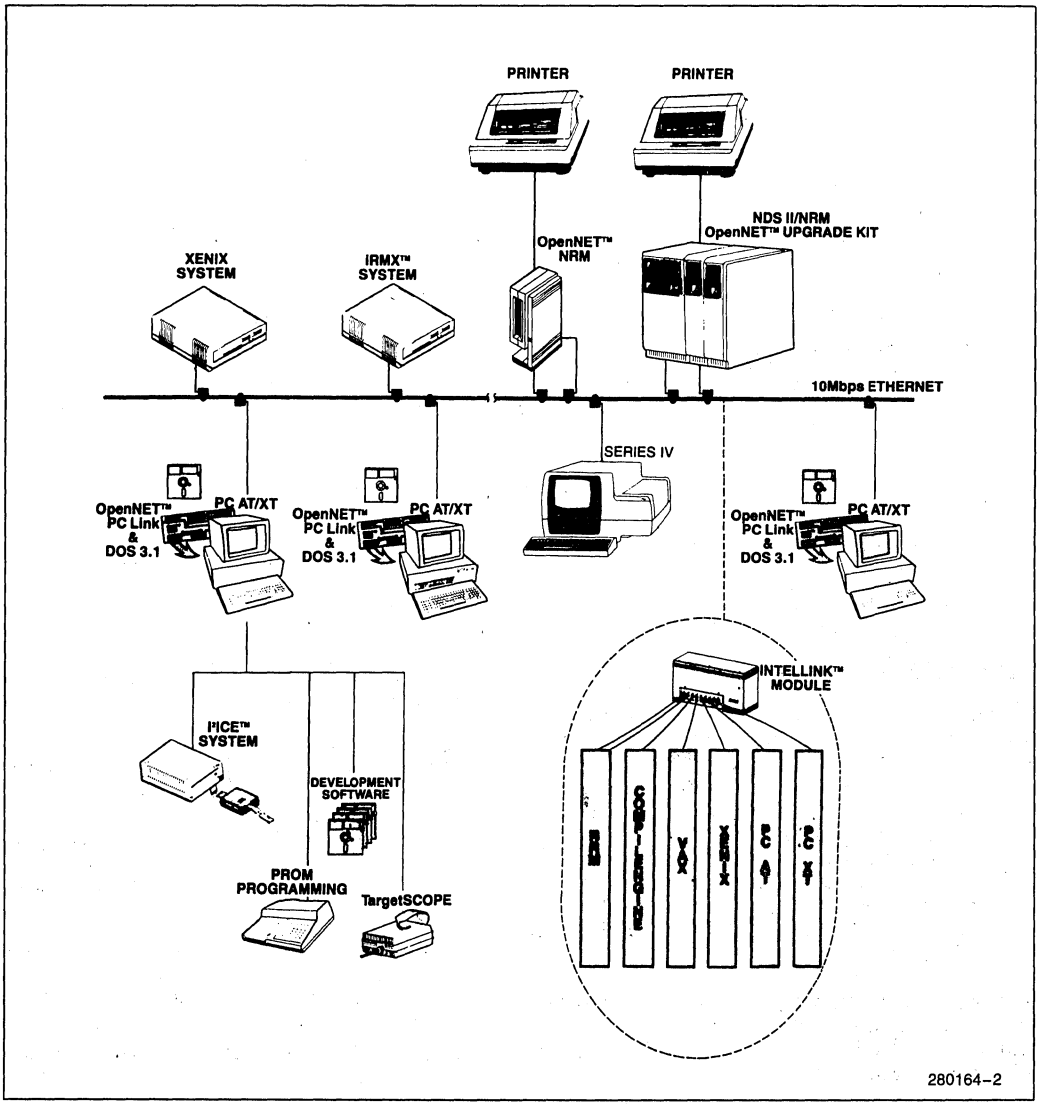


Figure 1. The OpenNET™ PC Link Environment

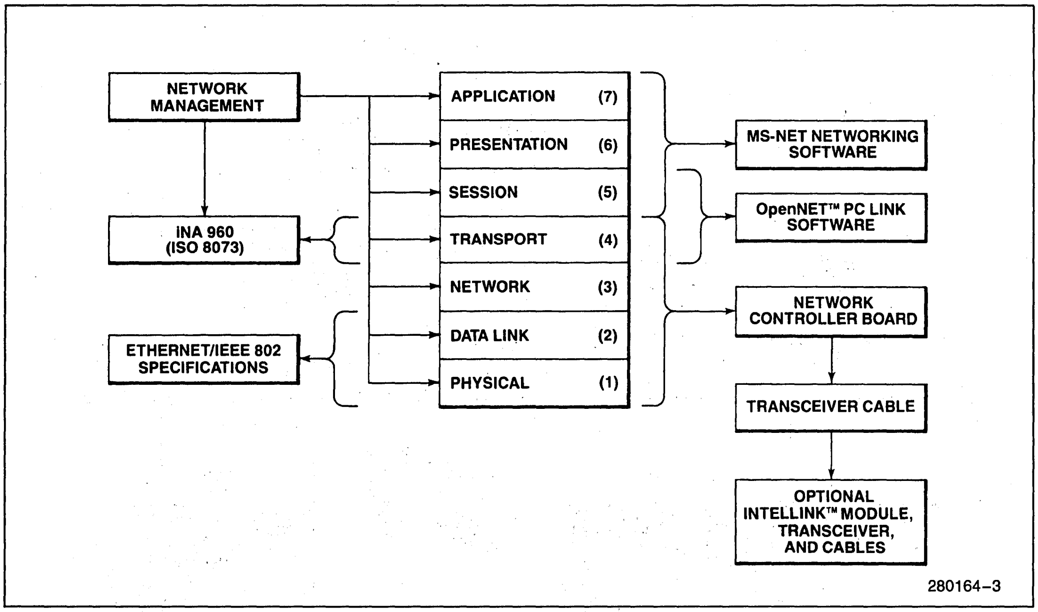


Figure 2. ISO/OSI OpenNET™ PC Link Implementation

PC SYSTEM REQUIREMENTS

For the PC system to function as a workstation on the OpenNET network, it must contain at least 192 KB of memory. A 32 KB memory window is shared between the PC system and the network controller board. The starting address of this window must be placed in an area of memory that does not conflict with the PC system's internal memory address space. The network controller board is jumpered at the factory to reflect a setting which is compatible with the PC system and most of the expansion boards available for use with the PC system.

In order for the network controller board and the OpenNET software to function properly, the PC system must use the DOS (MS-DOS or PC-DOS) operating system, version 3.1 or later.

FUNCTIONAL DESCRIPTION

The OpenNET PC Link enables a PC system to be configured as a consumer workstation in the OpenNET network environment. This enables a PC system to access and share files and remote printers on a remote file server. After establishing a connection with a remote server, the user can access different directories by connecting drive letters at the PC system to the desired directories.

Creating a PC System Consumer Workstation

The PC system is easily configured as a consumer workstation on the OpenNET network. The following steps summarize how to configure the PC system for use as a workstation:

- Install the OpenNET PC Link network controller board in the PC system and connect the PC system to an Ethernet transceiver or Intellink™ module.
- Configure the OpenNET PC Link software to reflect the name and network address assigned to the PC system and each remote server system that the PC system will access.
- Define the PC system user as a valid user of the remote server system.

To connect with and access remote resources over the OpenNET network, perform the following steps:

- Invoke the PC system's consumer networking software.
- Execute a connect-to-server command.
- Execute standard DOS commands.

The PC system user can now access remote resources (files, directories, or printers) at remote servers on the network.

The user has the option of automatically connecting to a remote server each time the DOS operating system is booted. This is done by placing networking commands in a DOS AUTOEXEC.BAT file.

Remote Server Access

The PC user gains access to a remote server by connecting an unused drive letter at the PC system to a remote home directory at the server. The server validates the PC system user by comparing the user name offered in the connect-to-server command with the server's user definition file. If the name is valid, the user is logged on to the server, and can access any file within the home directory. Multiple subdirectories may be created within the home directory. The user is restricted from accessing directories or files located above the user's home directory.

Transparent Access to Multiple Directories

A PC system user may access multiple directories at a file server. This is done by defining multiple users (giving users access to different directories) at the server. After establishing a connection to the server, the user can access different directories by connecting drive devices at the PC system to the desired directories.

Data and resource sharing are implemented via transparent remote file access. This enables the user to work with remote data files and resources residing at server systems on the network as if they were resident on the PC system. Users of a remote server may be given access to the same home directory, enabling multiple users to access and share remote data files. The access rights of remote data files can be changed to enable all or some of the users to read, write, or delete files in that directory.

Using DOS Commands Across the OpenNET Network

Once the PC system has been connected to a remote server on the network, almost any DOS command can be used with remote files and directories. The exceptions are commands that manage physical devices (e.g., FORMAT). The MS-NET software reports an error message if an invalid DOS command is sent across the network. Using DOS commands, the user can manipulate drives, files, and directories as follows:

- Look at and list remote directories and files.
- Copy files back and forth between a PC system and a remote server.
- Redirect print requests to a remote printer.
- Set and reset the read-only attribute of remote directories and files.
- Map drive assignments to remote directories.
- Set the path to remote directories and files.

Shared Printer Access

The PC system can be linked to a remote printer that is connected to a server on the OpenNET network. This enables the user to take advantage of the remote printer services, thus freeing the user from having to install a printer at the PC system.

A PC system user can print local or remote data files by first connecting the PC system's logical printer device to the remote server's printer spool. Then, the MS-NET networking software command NET PRINT is used to print the file on the remote printer device.

Table 1. MS-NET Networking Commands

Command	Description
APPEND	Locates a file which is outside the current directory.
NET CONTINUE	Restarts the disk redirector or print redirector programs.
NET HELP	Displays a help file with information about MS-NET commands.
NET NAME	Displays the name assigned to your PC system.
NET PAUSE	Temporarily halts the disk redirector and print redirector programs.
NET PRINT	Prints a file on a remote printer.
NET START REDIRECTOR	Invokes the OpenNET PC Link consumer networking software.
NET USE	Connects a device at the PC system to a remote server or printer.

Microsoft Networks Software

The Microsoft Networks (MS-NET) networking software is included as part of the OpenNET PC Link software. The MS-NET software manages the transfer of information between the PC system and a remote server. Once a connection is made between the PC system and a remote server or printer, the user uses the MS-NET software (in conjunction with DOS commands) to access and manipulate remote files or printers. Table 1 presents a list of MS-NET commands and a description of each command.

The MS-NET networking software displays a message each time a command is successfully completed. If an error is made, the software displays an error message listing the probable cause of the error and suggestions for correcting it. On-line help files enable the user to quickly reference MS-NET commands and obtain the correct syntax for entering commands.

OpenNET PC LINK SPECIFICATIONS

Host Requirements

IBM PC AT or PC XT computer system

- 192 KB of system memory
- DOS (MOS-DOS or PC-DOS) operating system, version 3.1 or later

Physical Characteristics

NETWORK CONTROLLER BOARD

Width: 13.315 in. (33.82 cm)

Height: 4.15 in. (10.54 cm)

Weight: 35 oz. (0.99 kg)

SOFTWARE

5¼ inch double-density disk (360 KB)

POWER REQUIREMENTS

+5V at 2.7 Amps

+12V at 0.5 Amps

Environmental Characteristics

Operating Temperature: 0° to 55°C (32° to 131°F)

Operating Humidity: Maximum of 90% relative humidity, non-condensing

Convection Cooling

Documentation

166664	OpenNET™ PC Link User's Guide

Optional Equipment

The following items can be ordered for use with the OpenNET PC Link:

PCLNK20F	Transceiver cable, 20 ft (6.1m)
PCLNK164F	Transceiver cable, 164 ft (50 m)
DCM911-1	Intellink module
iMDX3015F	Transceiver

ORDERING INFORMATION

PCLNK	OpenNET PC Link: Consists of a network controller board, a PC XT card support, software, and a user's guide

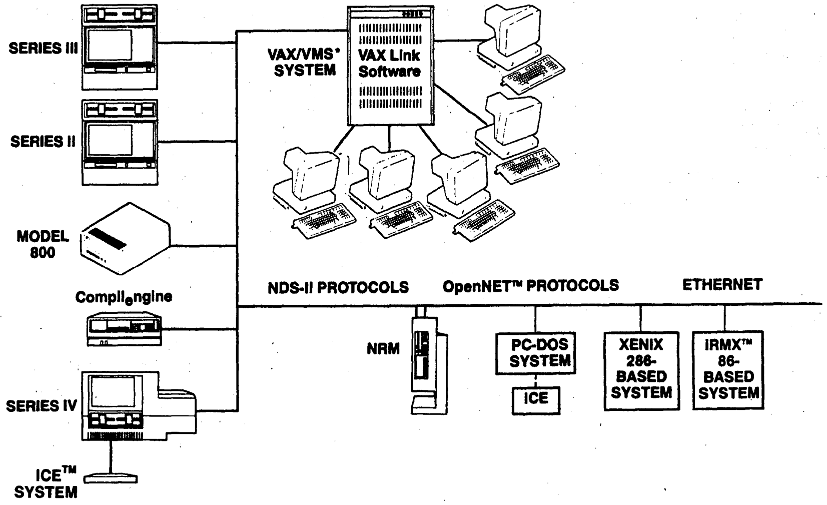
intel®

NDS-II/VAX* LINK NETWORKING SOFTWARE

- **Links VAX/VMS* to both NDS-II and OpenNET™ Development Environments**
- **Transfers Data via High-Speed Standard Ethernet/IEEE 802.3**
- **Enables File Transfer Between the NRM and VAX or MicroVAX II***
- **Offers VAX Users Access to All NRM File Services**
- **Optimizes Computing Resources with Distributed Job Control**
- **Authenticates User File Access Privileges for All Network Resource Manager (NRM) File Operations**
- **Requires a Digital Equipment Corporation DEUNA or DEQNA Communication Board for Operation (Not Supplied)**
- **Co-Exists with DECNET on the Same DEUNA or DEQNA Board**
- **Supports Multiple VAXs and Network Resource Managers (NRMs) in a Single Network**

NDS-II/VAX Link is an Ethernet-based communication link between Intel's Network Resource Manager (NRM) and a Digital Equipment Corporation (DEC*) VAX and MicroVAX II minicomputer. The NDS-II/VAX Link enables users to transfer files to/from the Series II/III/IV, Model 800, and the high-performance 286/310 based Compilengine. VAX users also have access to systems connected on the OpenNET network via the NRM. All data that is stored at the NRM is visible to, and can be accessed by, VAX users.

A major advantage of the NDS-II/VAX Link is its ability to optimize computing resources on the network via Distributed Job Control (DJC). DJC allows VAX users to queue jobs for remote execution upon the NRM. Similarly, NRM users can send jobs for remote execution upon the VAX. For example, CPU intensive jobs, such as compiles, can be sent from the VAX to idle Intel workstations for execution, saving valuable computational power for other activities. Or engineers using Intel development workstations can send special jobs to the VAX.



231299-2

Figure 1. NDS-II/VAX Link Enables High Speed Ethernet Data Transfers between the NDS-II, OpenNET™ and VAX Development Environments

NOTE:
All connections are on the same Ethernet cable.

*DEC, VAX, MicroVAX II and MicroVMS are trademarks of Digital Equipment Corporation.

August 1986
Order Number: 231299-003

NDS-II/VAX Link supports numerous commands that are initiated at the VAX. These commands are similar to Digital Command Language (DCL) commands and execute at the DCL command level. Users can obtain information on all commands by using the standard VMS* help facility. Commands cover general link operations (NVOPEN, NVCLOSE, NVLOGON, NVBYE, NVMESSAGE), distributed file system services (NVCOPY, NVDIRECTORY, NVCREATE, NVRENAME, NVDELETE, NVSET) and Distributed Job Control functions (NVCREATE/QUEUE; NVCREATE/IMPORT, NVEXPORT, NVCANCEL, NVSTATUS). Summaries of the more important commands follow below:

General Link Commands:

NVOPEN allows a VAX user with OPERATOR privilege to startup the link. NVCLOSE allows a VAX user to gracefully shutdown the link.

NVLOGON gives a VAX user access to the NDS-II files on a given NRM. The user must provide an NDS-II username and password. NVBYE logs a user off from a given NRM.

Distributed File System Commands:

NVCOPY copies a single file or a group of files from the VAX to the NRM or vice versa. The command accepts wildcard filename specifications and supports common sequential VAX/VMS file types.

NVDIR lists the directory entry of the NRM file(s) specified. The directory listing is in iNDX format. The user can request an expanded directory listing consisting of the filename, owner name, length, type, and owner and world access rights.

NVDELETE deletes one or more files or directories from the NRM file system. The invoker must have DELETE permission on each file or directory specified. A directory must be empty before it is deleted.

NVCREATE creates a new directory in the NRM file system. The invoker must have write access to the parent directory where the new directory is being created.

NVSET displays and/or changes the protection mask for files in the NRM file system. The user must have the appropriate access rights to the files in question when using the NVSET command.

Distributed Job Control Commands:

NVCREATE/QUEUE creates NRM queues. Queues must be created before they are used by either NVCREATE/IMPORT or NVEXPORT.

NVCREATE/IMPORT creates an import station on the VAX that serves the specified existing NRM queue. This is a privileged command that can only be executed by users having OPERATOR privilege.

NVEXPORT queues a job for execution in the NRM queue specified in the command parameters. The exported job will be executed by an import station serving the specified queue.

NVSTATUS lists NRM queues, the number of jobs waiting in the queues, and the number of import stations serving the queues. By specifying the /FULL qualifier the user can display detailed information on each job in the queue(s).

NDS-II/VAX Link supports up to 16 users on the link at a given time. With multiple users the link is operated in time-sharing fashion thus, giving each user the appearance of a dedicated connection to the NDS-II.

NDS-II/VAX Link also supports multiple VAXs and multiple NRMs in a single network. Users on separate VAXs can access the same NRM simultaneously, and users on the same VAX can access different NRMs. Multiple NRM support is only supported under VAX/VMS* version 4.0 (and later versions). Separate NDS-II/VAX Link software licenses must be purchased for each VAX connected in a multiple VAX/NRM environment.

NDS-II/VAX Link requires a DEC DEUNA-AA or DEQNA communication assembly that must be purchased from and installed by DEC. In addition, a standard external Ethernet transceiver cable is required to connect the DEUNA or DEQNA assembly to the Intel NDS-II Intellink™ Module.

SPECIFICATIONS

Operating Environment:

REQUIRED HARDWARE:

NDS-II NRM or OpenNET NRM

DEC* VAX 11/730, 11/750, 11/780, 11/782, or 11/785, or MicroVAX II Minicomputer

DEC DEUNA-AA or DEQNA Assembly (from DEC)

Ethernet Transceiver Cable (Intel iMDX-457 or equivalent)

REQUIRED SOFTWARE:

iNDX Network Operating Software, Version 3.0 or later

VAX/VMS* or MicroVMS* Operating Software, Version 4.2 or later

Documentation:

NDS-II/VAX Link User's Guide (Order Number 122301-002)

Software Support:

This product includes a 90-day initial support consisting of subscription services and telephone hotline support. Additional software support services are available separately.

Future Update Kits are not covered under warranty and must be purchased separately.

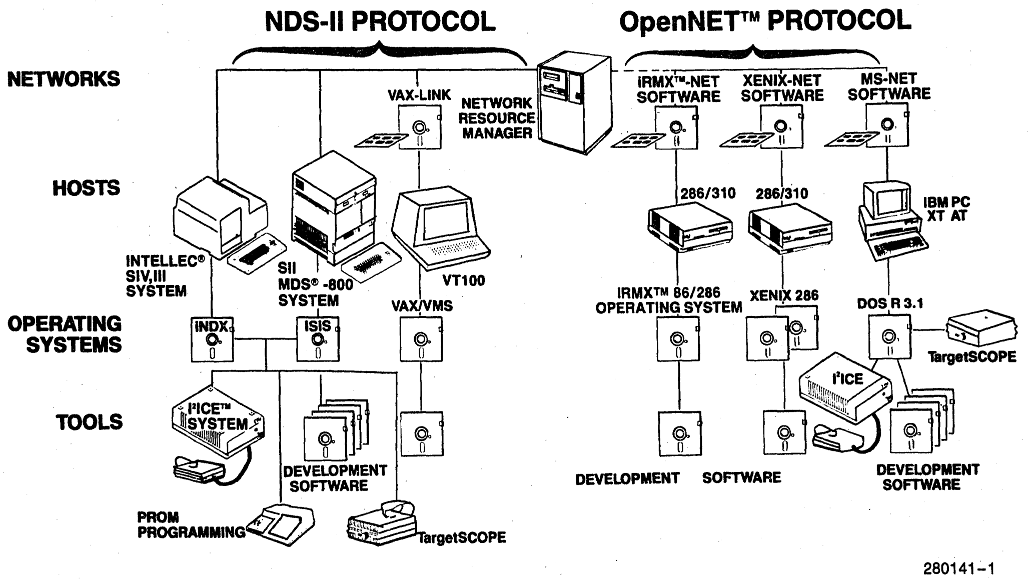
ORDERING INFORMATION

Part Number	Description
iMDX392	NDS-II/VAX* Link 9 track magnetic tape media
iMDX 393F	NDS-II/MicroVAX II Link RX 50 5¼" Floppy Media
iMDX 393T	NDS-II/MicroVAX II Link TK 50 Cartridge-tape Media

iMDX 555
NDS-II NRM OpenNET™ UPGRADE

- **Provides Series II/III/IV, Model 800 and VAX*/VMS Development Customers an Upgrade Path into the OpenNET™ Networking Environment**
- **NDS-II NRM now Becomes a File Server for OpenNET Users Including XENIX, iRMX™ 86, and PC-DOS Systems**
- **Supports Large Number of OpenNET Workstations up to the Physical Connection Limit for Ethernet (100 Direct Connections via Transceivers or over 800 via Intellink™) Boxes with the Ability for 30 to Simultaneously Access the NRM**
- **Transparent File Access to the NDS-II NRM from XENIX*, iRMX 86, and PC-DOS Systems**
- **Additional OpenNET Workstations Can Be Added without Reconfiguring the Network**
- **Authenticates OpenNET User File Access Privileges for all NRM Resources**
- **Shared Resources, e.g., Spooled Line Printer, up to 336 MB Winchester Central Disk Storage, Tape Archive**
- **Uses ISO 8073 Transport and Ethernet/IEEE 802.3 Standard Communication Protocols**

The NDS-II NRM OpenNET Upgrade contains software and hardware that allow the Network Resource Manager (NRM) to function as a file server in an OpenNET network environment. The Intellec® Series II/III/IV and Model 800 development systems, and Digital Equipment Corporation's VAX minicomputers can now share files residing on the NRM with XENIX, iRMX 86, and PC-DOS systems. XENIX, iRMX 86, and DOS system users can also take advantage of the NRM resources such as the spooled line printer, tape archive, and fast disks.



280141-1

NDS-II NRM Links Series II/III/IV and VAX/VMS to iRMX™ 86, XENIX, and DOS Systems Via OpenNET

*VAX is a registered trademark of Digital Equipment Corporation
*XENIX is a trademark of Microsoft
*IBM is a trademark of International Business Machines

October 1986
Order Number: 280141-002

FUNCTIONAL DESCRIPTION

The NDS-II NRM OpenNET Upgrade provides the capability for existing NDS-II users to expand into the OpenNET networking world. OpenNET network file access is based on protocols developed jointly by Intel, Microsoft, and IBM* to interconnect systems running different operating systems. This includes systems running XENIX, iRMX 86, PC-DOS, VAX/VMS, and now iNDX (on the NRM) operating systems. Alternatively, new NDS-II owners can integrate development tools running on XENIX, iRMX 86, and PC-DOS workstations, with the NRM file server.

These capabilities are achieved by combining the iNDX OpenNET software and OpenNET transport engine—the iSXMTM 552 board. The iSXM 552 board implements the industry standards ISO 8073 transport protocol (iNA 961) and Ethernet/IEEE 802.3 physical data link technology.

The NDS-II and OpenNET networks utilize separate communication boards (i.e., iSBC® 550 and iSXM 552 boards, respectively); therefore, there is no contention for communication resources.

Some limitations (e.g., formatting remote disks from a local workstation) do apply to iNDX OpenNET. See iNDX OpenNET User's guide for specific command support.

Number of Users Supported on the OpenNETTM Network

Each time an OpenNET workstation connects or "logs on" to the NRM (e.g., for file or printer access), a virtual circuit is created between the workstation and the NRM. iNDX OpenNET supports up to 30 simultaneous NRM users (virtual circuits) on the OpenNET side (one virtual circuit per OpenNET consumer node). When that limit is reached, no new circuit will be established until one of the existing circuits is closed. This means that although the number of physical OpenNET workstations on the network is potentially the limit of Ethernet connections (e.g., 100 via transceivers or over 800 via cascaded Intellink boxes), only 30 can access the NRM at any given time.

Because of this limit, iNDX OpenNET implements a least-recently-used algorithm to maximize virtual circuit availability. An idle OpenNET consumer (no outstanding file requests or log-ons) is automatically disconnected when the 31st network connection request is made. Alternatively, a workstation that is turned off without disconnecting from the NRM is automatically disconnected approximately 10 minutes after it is turned off.

Under normal file usage, the number of users on the OpenNET side should not affect the number of users on the NDS-II side (i.e., Series II/III/IV, VAX/VMS etc.)

NRM User Creation/File Access

The NDS-II NRM OpenNET Upgrade allows OpenNET workstations to transparently access files on the NRM without physically copying those files onto local disk. Files to be shared between NDS-II workstations (e.g., the Model 800, Series II/III, Series IV) and OpenNET workstations (e.g., PC-DOS, XENIX, iRMX) reside on the NRM.

Every OpenNET user is created on the NRM console with two simple commands:

- USERDEF DEFINE gives each user a unique logon ID and home directory (it is mandatory to specify a home directory)
- CHPASS creates the user password.

These two commands allow the NRM Administrator (i.e., SUPERUSER) to control access by other OpenNET users to files as well as other resources (e.g., print spooler) residing on the NRM. Each time an OpenNET user attempts to access an NRM resource, an automatic log-on process occurs. Those users who do not have the proper log-on ID and password will be denied access to the requested NRM resource.

For file sharing at the NRM, users can be created with the same home directory. Then, the access rights of the home directory can be changed to allow all the users to read, add or delete files in that directory. The iNDX hierarchical protected file system supports owner and world access rights to files (READ, WRITE and DELETE) and directories (DISPLAY, ADD-ENTRY, and DELETE). These access rights may be set by the owner of the file/directory, or by the SUPERUSER (on the NRM). The SUPERUSER on the NRM terminal has access rights to all resources at the NRM as well as the authority to create and delete users.

iNDX supports concurrent read access to files and single write access to files (i.e., while a file is opened for writing by a user, no other user can read from or write to that file).

OpenNETTM Consumers

To become a valid network user under OpenNET, the user configures his/her OpenNET workstation as an NRM OpenNET consumer. This involves performing three simple steps at the workstation:

- Adding the NRM's communication address in the workstation's session data base.
- Activating the workstation's consumer network software
- Executing a connect-to-NRM command

This establishes connection between the workstation and the home directory specified during user creation at the NRM (with the USERDEF DEFINE command).

A user may access different directories residing on the NRM. This can be done by defining multiple NRM users (giving access to different directories) at the NRM console. After establishing a connection with the NRM, a user at that workstation can access different directories with different log-ons.

NRM Print Spooler Access

An OpenNET user can print files on the NRM by first connecting the workstation's printer logical device to the NRM print spooler (with the NET USE command). Then, local and/or network files are printed by using the workstation's local network print command.

For example, a DOS user would enter the NET PRINT command from the DOS workstation. A XENIX user, on the other hand, would simply use the RPRINT command with the specified NRM destination to print a file at the NRM. Alternatively, the XENIX user can use the copy command (CP) to copy local print files to the NRM print spooler (:SP:).

Once a file has been sent to the NRM print spooler, that print job can be cancelled with the DELETE command at the NRM terminal by the network administrator.

The OpenNET™ Communications Engine (iSXM 552 Board)

The iSXM 552 board integrates a high performance processor, the 80186, and a powerful Local Area Network (LAN) coprocessor, the 82586, on a single MULTIBUS® board. With the iNA 961 transport software, this solution set provides the implementation of the first 4 layers of the ISO OSI (7-layer) network communications model.

This Intel LAN solution offers reliability and easy expandibility. iNA 961 provides internal detection and correction of communication mediums and workstations so that a malfunction at a given point will not cause total network failure. Moreover, stations can be added or deleted from an existing network without reinitialization or reconfiguration of all other workstations.

OpenNET™ and Interconnection Hardware

Although the OpenNET connection requires a communications board (iSXM 552) different from NDS-II (DFS/ISIS) communications boards (iSBC 550 board), all existing Intellink™ boxes, transceivers, and cables remain the same. Furthermore, the same (one) Intellink can be used to connect Series II/III/IVs, VAXes, and OpenNET workstations, as long as cables from both the iSBC 550 and iSBC 552 boards are connected to that Intellink box. The Series II/III/IV's, VAXes will only respond to the messages from the iSBC 550 board while the OpenNET workstations will only understand the messages from the iSBC 552 board. This helps the user maximize usage of current interconnect hardware.

SPECIFICATIONS

Hardware Supplied

iSXM 552A OpenNET communication board
iSBC028A 128K RAM board
Internal cable assembly

Software Supplied

iNDX OpenNET software
iNDX Operation System

Operating Environment

- Hardware required:
  NDS-II NRM with 2 unoccupied MULTIBUS board slots
- OpenNET workstation connection requirements
  Personal Computer:

PCLNK	OpenNET PC Link hardware and software kit to connect the PC XT, PC AT, and compatible systems to the NRM via the OpenNET network; requires DOS 3.1 or higher

XENIX System:

XNXNETNRIKIT	OpenNET-XenixNET, iNA 961, iSXM 552 and Xenix-NET Pass-through Kit

RMX System:

RMXNETKITWRI	iRMX Networking Software for a 286/310 system running the iRMX 86 R6.0 Operating System to connect to the NRM via the OpenNET network, SXM 552S.

VAX or MicroVAX running VMS:
—VAX OpenNET server s/w and controller board

- Documentation:

  NDS-II OpenNET User's Guide (order no. 138809-001)

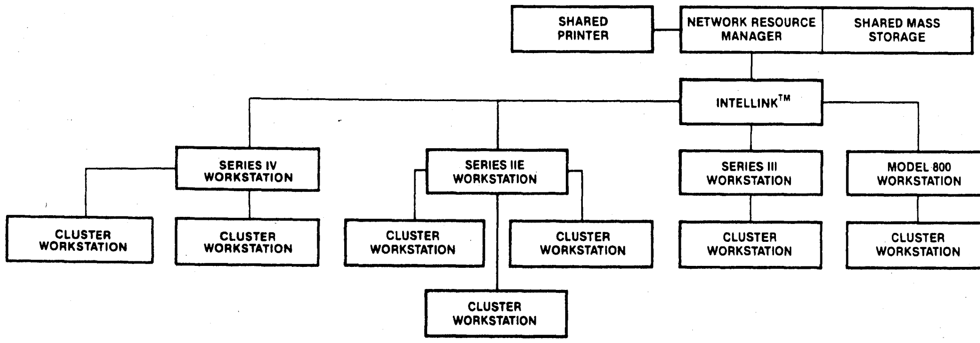
ORDERING INFORMATION

Part Number	Description
iMDX 555	NDS-II NRM OpenNET Upgrade Package

iMDX-581
ISIS CLUSTER BOARD PACKAGES

- **Converts Spare Slots in Series II, III, IV, or Model 800 Workstations into Additional Workstations**
- **Up to Seven Additional NDS-II Workstations May Reside in One Development System Host**
- **Utilizes the Powerful ISIS-III(C) Operating System**
- **Supports 16-bit Development with Local ASM-86 and PL/M-86, and Via NDS-II Distributed Job Control**
- **Supports 8-bit Macroassemblers and High-Level Languages**
- **Supports all 8-bit ISIS-Based Software Development Tools Including the AEDIT-80, Text Editor, Program Management Tools, and NDS-II Electronic Mail**
- **Provides Execution Environment for 8085-Based Application Programs**
- **Compatible with a Variety of 9.6K or 19.2K Baud Terminals**

The ISIS Cluster Board Package is an NDS-II upgrade that cost effectively supports incremental software workstations on the network. Each Cluster board provides an 8085 CPU, 4K of ROM and 64K of RAM, and must reside in a Series II, Series III, Series IV, or Model 800 development system host. When attached to a user-supplied terminal, an ISIS Cluster workstation will boot onto the NDS-II and provide an ISIS environment which allows users to log on to the network and run Intellec®-supported 8-bit software, as well as "export" jobs to other network resources.



231408–1

Figure 1. Example of an NDS-II Configuration

December 1985
Order Number: 231408-001

FUNCTIONAL DESCRIPTION

Summary: The ISIS Cluster board is a single-board computer centered around an 8085AH-2 CPU running at 4.0 MHz. 64K bytes of dual-ported RAM are provided on-board, along with 4K of ROM preprogrammed with a bootstrap program and self-test diagnostics.

The ISIS Cluster MULTIBUS® interface provides data and address interface latches. The serial I/O interface provides a full duplex RS232C serial data communications channel that can be programmed to handle serial data transmission at 19.2K or 9.6K baud. Software reset may be accomplished using the BREAK key on the terminal.

A block diagram of the ISIS Cluster board is shown in Figure 2.

Central Processing Unit

Intel's powerful 8-bit 8085AH-2 CPU running at 4.0 MHz is the central processor for the Cluster board. It is fully software compatible with all 8-bit ISIS-based languages and utilities which run on the Intellec Model 800, Series II/80, Series II/85, or Series IIE.

System ROM

4K bytes of non-volatile read only memory are included on the Cluster board using Intel's 2732A EPROM. Preprogrammed with the ISIS Cluster Boot program, the system ROM provides boot-up and diagnostic capabilities, and a generalized I/O system.

The Boot program communicates with the operator via an interactive console. Upon reset of the Cluster system, execution is handled by the bootstrap PROMs which overlay 4K bytes of system RAM, initialize Cluster board devices, run self-test diagnostic, and perform a communication handshake before prompting the user.

RAM

The Cluster board uses eight 2164 RAMs and a dual port RAM controller to provide 64K of dual-ported dynamic read/write memory. Slave RAM decode logic allows extended MULTIBUS addressing with a 1 Megabyte address space, so that RAM accesses may occur from either the Cluster board or from the network communication boards interacting via the MULTIBUS interface. Since on-board RAM accesses do not require MULTIBUS accesses, the bus is available for other concurrent operations. Dynamic RAM refresh is accomplished automatically by the Cluster board.

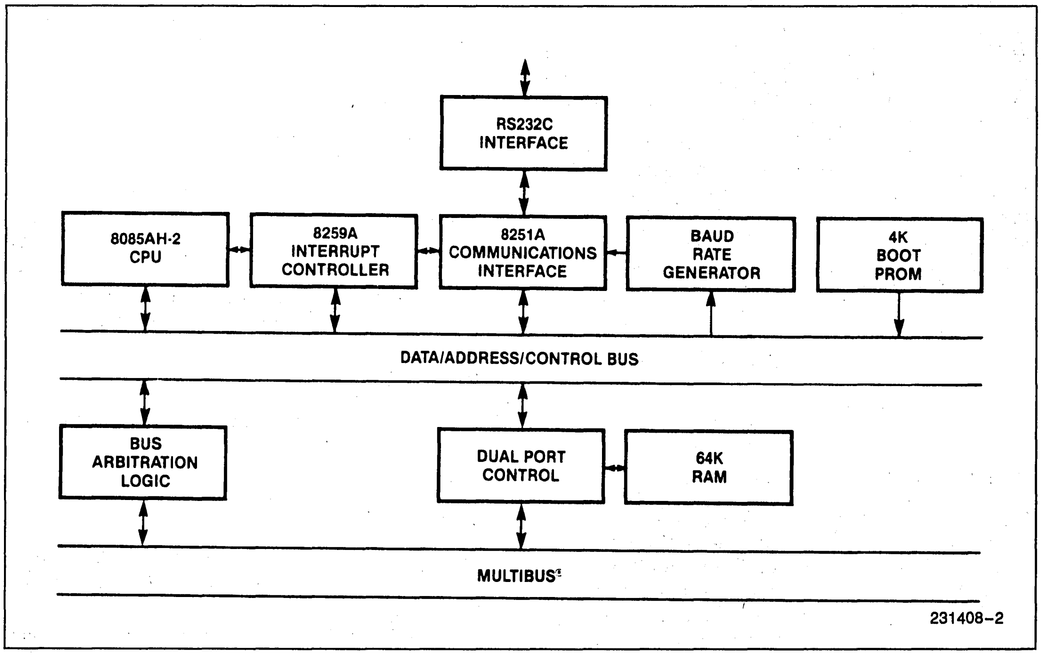


Figure 2. Block Diagram of the ISIS Cluster Board

Serial I/O

A programmable communications interface using the Intel 8251A USART (Universal Synchronous/Asynchronous Receiver/Transmitter) is on the Cluster board, and provides a full duplex RS232C serial communications channel. The transmit and receive lines are link exchangeable to enable a data set or data terminal to be used with the Cluster board. The board is pre-set for 9600 baud, but may be jumpered for 19.2K baud.

Programmable Timers

The interval timer capability is implemented with an Intel 8254 Programmable Interval Timer. The 8254 includes three 16-bit BCD or binary counters. The first two counters are not used. The output from the third counter is applied to the serial I/O interface and provides the baud rate frequency for serial communications.

Interrupt Controller

The Cluster board also includes an Intel 8295A Interrupt Controller. It is pre-configured with Interrupt 1 triggered by the BREAK key on the user-supplied terminal.

MULTIBUS® Interface

The Cluster board is a complete computer on a single board, capable of supporting a variety of 8-bit development tools. For applications requiring additional processing capacity, the Cluster board provides full MULTIBUS arbitration control logic. The bus arbitration logic operates synchronously with a MULTIBUS clock. All memory references made by the CPU refer to the on-board RAM. The Cluster board cannot access devices local to the host development system, but all of the shared network resources are accessible.

The Cluster board communicates with the Network Resource Manager via the MULTIBUS interface and the network communication board set in the host development system.

System Configuration

Each ISIS Cluster board requires one master slot in an Intellec cardcage. The host development system may be a Model 800, Series IV, Series II or IIE, or Series III or IIIE with an optional expansion chassis. A Series II or IIE with an expansion chassis will support a maximum of seven ISIS Cluster workstations, since the Integrated Processor Card and Network Communication boards occupy three of the ten cardcage slots. A Model 800 will support a maximum of 2 ISIS Cluster workstations, and Series IV workstation will support a maximum of 4 ISIS Cluster workstations. Each ISIS Cluster workstation counts as one additional network workstation, so the maximum number of Cluster workstations on a network is constrained only by the total number of users supported by the NDS-II Network Resource Manager. NDS-II iNDX Release 2.8 or later will support ISIS Cluster workstations in any Intellec development system host, including the Series IV.

Programming Capability

The Cluster workstation's ISIS environment supports all 8-bit Intellec-supported ISIS-based software, including the programmer-oriented AEDIT-80 text editor, PMT-80 Program Management Tools, NDS-II Electronic Mail, 8-bit macroassemblers, and PL/M, FORTRAN, PASCAL, and BASIC high-level 8-bit languages. 16-bit development is supported by the ASM86 cross assembler and the PL/M-86 cross compiler, or by "exporting" any 16-bit job to a 16-bit workstation for execution.

SPECIFICATIONS

CPU: 4.0 MHz 8085AH-2

MEMORY:

On-board RAM, 64K bytes, dual-ported
On-board ROM, 4K bytes preprogrammed with the ISIS Cluster Bootstrap Program

Interfaces

SERIAL I/O:	RS232C compatible, programmable interface
BUS:	MULTIBUS compatible, TTL level
TIMER:	3 programmable 16-bit BCD or binary counters, 1 used as baud rate timer
INTERRUPTS:	1 interrupt level available to user via the BREAK key on the terminal

Physical Characteristics

Two-sided printed circuit board fits into Intellec cardcage:

Length:	12 inches
Width:	6.75 inches
Depth:	0.062 inches

Internal flat ribbon cable connects ISIS Cluster board edge connector to the development system rear panel.

External 10-foot RS232C compatible cable connects the development system rear panel to a user-supplied terminal.

Electrical Characteristics

DC Power Requirements (from Mainframe)

V_{CC} = +5V, 4.5 Amps
V_{DD} = +12V, 25 mA
V_{AA} = −12V, 23 mA

Environmental Specifications

Operating Temperature: 0°C to 55°C
Humidity: up to 90%, without condensation

Documentation

iMDX-581 Installation, Operation, and Service Manual (#122293)

NDS-II ISIS-III(C) User's Guide Supplement (#122098)

Equipment Required

Recommended Terminals* (one per ISIS Cluster Board)

The following terminals have been tested and found to be interface compatible with the ISIS Cluster board; configuration files are provided for these terminals. Customers are advised to select terminals to meet their own environmental specifications.

Hazeltine, Model 1510
Televideo, Model 910+, 925, 950
Lear Seigler, Model ADM 3A
Adds Viewpoint, Model 3A+
Qume, Model 102
Zentec, Model ZMS-35, Cobra

*All of the recommended terminals run at 9.6K or 19.2K baud.

CAUTION: Other RS232C-compatible devices may not meet Intel environmental specifications, and could degrade overall system performance.

Host Development System (requires one open 6.75 x 12 in. master slot in system cardcage per ISIS Cluster board):

Series II/85 or Series IIE*
Series III or Series IIIE*
Model 800**
Series IV
*with optional Expansion Chassis
**supports maximum of 2 ISIS Cluster Boards

Workstation Upgrade Kit (one per host system):

iMDX-455 for Series II, III, or Model 800
iMDX-456 for Series IV

NDS-II Network Resource Manager with Hard Disk Mass Storage

Software Required

For Series II, III, or Model 800 Host:

NDS-II iNDX Operating System, Release 2.0 or later

ISIS-III(N)/III(C) Operating System, Version 2.0 or later*

For all Development System Hosts, Including Series IV:

NDS-II iNDX Operating System, Release 2.8 or later

Series IV iNDX Workstation Operating System, Release 2.8 or later**

ISIS-III(N), version 2.2 or later**

ISIS-III(C), version 2.2 or later**

*included with NDS-II Release 2.0
**included with NDS-II Release 2.8

ORDERING INFORMATION

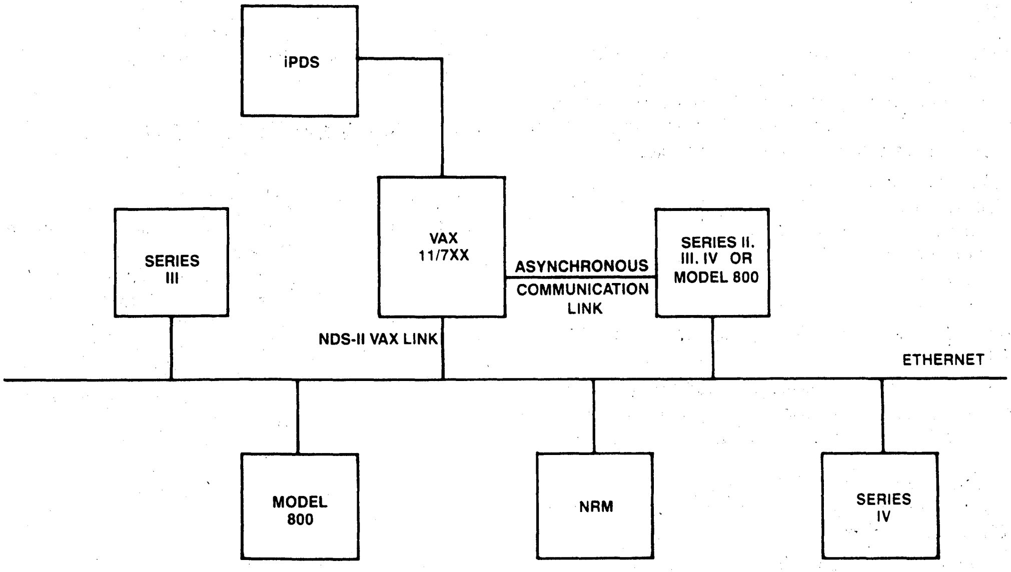
Part Number	Description
iMDX-581	ISIS Cluster Board Package for Series II, Series III, Series IV, or Model 800—includes processor board, cables, and documentation. Must be installed on NDS-II in a Model 800, Series II, Series III, or Series IV workstation and connected to a user-supplied terminal.

INTEL ASYNCHRONOUS COMMUNICATIONS LINK

- Communications Software for VAX* Host Computer and Intel Microcomputer Development Systems
- Compatible with VAX/VMS* and UNIX† Operating Systems
- Supports Intel's Model 800, Intellec® Series II, Series III, Series IV and iPDS™ Microcomputer Development Systems
- Supports NDS-II Workstations
- Allows Development System Console to Function as a Host Terminal
- Operates through Direct Cable Connection or over Telephone Lines
- Software Selectable Transmission Rate from 300 to 9600 Baud

Intel's Asynchronous Communications Link (ACL) enables Intel microcomputer development systems to communicate with a Digital Equipment Corporation VAX family computer. The link supports Intel Model 800, Intellec Series II, Series III, Series IV or iPDS development systems. Programmers can use the editing and file management tools of the host computer and then download to the Intel microcomputer development system for debugging and execution. Programmers can use their microcomputer development system as a host terminal and control the host directly without changing terminals.

NDS-II Example



210903-1

NOTE:
NDS-II VAX Link is an Ethernet Link between VAX*/VMS and Intel's Network Resource Manager. This product is also available from Intel.

*VAX and VAX/VMS are trademarks of Digital Equipment Corporation.
†UNIX is a trademark of Bell Laboratories.
††VADIC is a trademark of Racal-Vadic Inc.

October 1986
Order Number: 210903-003

FUNCTIONAL DESCRIPTION

The Asynchronous Communications Link (ACL) consists of cooperating programs: one that runs on the VAX computer, and others that run on each microcomputer development system. The development system programs execute under the ISIS-II or ISIS-III(N), ISIS-IV, ISIS-II(W), or ISIS-PDS operating system. They invoke the companion program on the VAX-11/7XX, which runs under either the VAX/VMS or UNIX operating system.

The link provides three modes of communication: on-line transmission, single-line transmission, and file transfer. In on-line mode, the development system functions as a host terminal, enabling the programmer to develope programs using the host computer's editing, compilation, and file-management tools directly from the development system's console. Later, switching to file transfer mode, text files and object code can be downloaded from the host to the development system for debugging and execution. Alternatively, files can be sent back to the host for editing or storage. In single line mode, the programmer can send single-line commands to the host computer while remaining in the ISIS environment.

The user can select transmission rates over the link from 300 to 9600 baud. The link transmits in encapsulated blocks. The receiver program validates the transmission by checking record-number and checksum information in each block's header. In the event of a transmission error, the receiving program recognizes a bad block and requests the sender to retransmit the correct block. The result is highly reliable data communications.

SOFTWARE PACKAGE

The Asynchronous Communications Link Package contains either a VAX/VMS or UNIX compatible magnetic tape, a single 8", double 8", Series-IV 5¼", and PDS 5¼" diskette compatible with the Intellec development system, and the *Asynchronous Communications Link User's Guide* containing installation, configuration and operation information.

HARDWARE CONNECTION

The Link sends data over an RS232C cable. The communication line from the host computer connects directly to a development system port.

TELECOMMUNICATIONS USING THE LINK

The ACL is ideal for cross-host program development using a commercial timesharing service. This configuration requires RS232C compatible modems and a telecommunications line. Depending on the anticipated level of usage, wide-area telephone service (WATS), a leased line, or a data communications network may be chosen to keep operating overhead low.

NDS-II ACCESS USING THE LINK

The ACL is ideal for interconnecting VAX host computers with NDS-II. This configuration requires that an NDS-II workstation be connected to the VAX host computer using the RS232C interface and to NDS-II using the Ethernet interface.

All three modes of communication operate identically on NDS-II. In the on-line mode, the development workstation operates as a host terminal, and concurrently, as an NDS-II workstation. It is an easy transition between the VAX and ISIS operating system environments as LOGON/LOGOFF sequences are not required to re-enter environments.

In file transfer mode, text and object files can be transferred from the VAX directly to the Winchester Disk at the NRM without first copying the files to the workstation local floppy disk. Similarly, files residing on the NDS-II Network File System (the Winchester Disk at the NRM) can be transferred directly to the VAX without using local workstation storage.

Using the EXPORT/IMPORT mechanisms of NDS-II, a network workstation which is not directly connected to the VAX can cause files to be transferred between the VAX and NRM. For example, any NDS-II workstation can "EXPORT" ACL commands to another "IMPORT"ing NDS-II workstation which is physically connected to a VAX. The "IMPORT"ing workstation executes the ACL command file causing the desired action to occur.

VAX ACCESS USING THE LINK

Users who want multiple workstations concurrently operating as VAX terminals (ONLINE mode) must physically connect each workstation to the VAX. However, users who want multiple workstations to be able to upload/download files, for example, must only physically connect one workstation to the VAX. By using the EXPORT/IMPORT mechanism of NDS-II as described above, the user can have multiple workstations accessing the VAX using only one connection.

SPECIFICATIONS

Software

Asynchronous Communications Link development system programs

VAX/VMS or UNIX companion program

Media

Single- or double-density ISIS 8" and Series-IV, iPDS 5¼" compatible diskette

600-ft. 1600 bpi magnetic tape, VAX/VMS or UNIX compatible

Data Transfer Speeds

All systems up to 9600 bps

Online Terminal Mode Speeds

Series II, Series III, Series IV — 2400 bps max PDS — 9600 bps max

Model 800 — equal or less than the Terminal speed

Manual

Asynchronous Communications Link User's Guide
Order No. 172174-001

Required Host Configuration

VAX-11/7XX running VAX/VMS (Version 4.1 and later) or fourth Berkley distribution of UNIX 4.2

Required Intel Development System Configuration

Model 800, Series II, Series III, Series IV, or iPDS under ISIS

Required Connection

RS232C compatible—cable 3M-3349/25 or equivalent; 25-pin connector 3M-3482-1000 or equivalent

Recommended Modems for Telecommunications

300 baud—Bell 103 modem; VADIC†† 3455 modem or equivalent

1200 baud—Bell 202 modem; VADIC 3451 modem or equivalent

9600 baud—Bell 209A (full duplex, leased line) or equivalent

NOTE:
Since one of the two Model 800 ports uses a current loop interface, Model 800 users need a terminal or modem that is current loop compatible, or a current loop/RS232C converter.
The model 800 might require modification by a qualified hardware technician. Intel does not repair or maintain boards with these changes.

ORDERING INFORMATION

Product Name

Asynchronous Communications Link

Ordering Code‡

iMDX 394 for VAX/VMS systems

iMDS 395 for UNIX systems

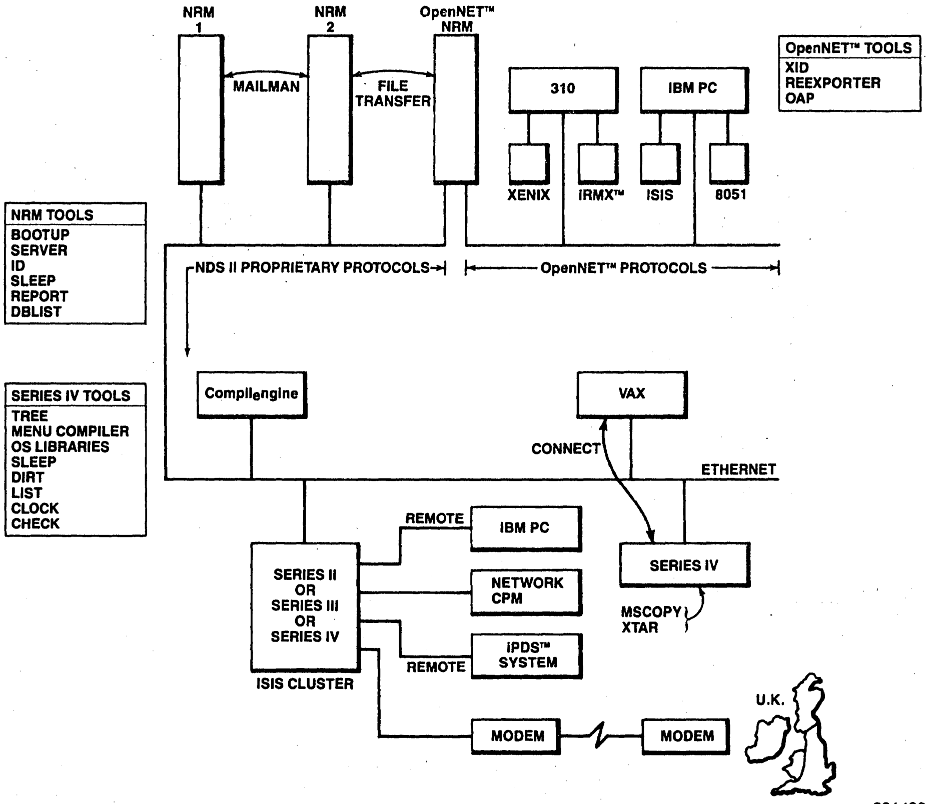
‡ See price book for proper suffixes for options and media selections.

NDS-II/Series-IV/OpenNET™ Toolbox

- Multiple NRM Communication
- Remote Series-IV from VAX* Terminal
- Series-IV Menu Compiler
- MS-DOS*/Series-IV Disk Read Utility
- XENIX Services for Any Workstation that can Access an OpenNET™ NRM
- Access to NDS II DJC for OpenNET™ Workstations
- Allow 8080 Based Intel Tools on 8086 Family Systems

The NDS-II/Series-IV/OpenNET Toolbox is a software only product that contains valuable collection of tools developed for the NDS-II, Series-IV and OpenNET user. These tools have been designed to make hybrid development system environments work together and to more fully automate the software developer's task. Many tools are provided with source to allow the engineer to customize these products to their own environment.

Note: However, this is not a supported product.



231488-2

Example of the Many Possible Connections Available with NDS-II/Series-IV/OpenNET™ Toolbox

*MS-DOS is a trademark of Microsoft Corporation
*VAX is a trademark of Digital Equipment Corp.
CP/M® is a registered trademark of Digital Research Inc.

May 1986
Order Number: 231488-002

CONNECT

CONNECT allows software developers to use their VAX terminal as a virtual terminal for their Series-II or Series-IV work station. Software developers can now run PSCOPE, ICE™ and I²ICE™ emulators from their VAX terminal, eliminating the need to switch terminals when debugging a program. This serial communications based program runs at 9600 baud for the Series-II and 2400 baud for the Series-IV. Complete support of the Series-IV menu line is available on the VAX terminal. CONNECT does not provide file transfer capability, this is provided for in either the VAX Link or ACL products. A separate serial cable, not supplied with Toolbox, is required for connecting the development system to the VAX. Source and generation are provided.

NRM to NRM Communications

The NRM to NRM communications package provides file transfer and printer spooling from one NDS-II network to another via ethernet. Two new commands are provided, NNCOPY and NNDIR, for Series-IVs running iNDX version 2.5 or greater. These commands do not function on the MDS-800 development system, ISIS Cluster, Series II, or Series III; although an ISIS work station may use export to run NNCOPY or NNDIR remotely. Full file protection is provided by this application. This product also requires that the NRM terminal run the slave program, NNL. The system administrator can prevent access to the NRM from remote systems by not executing NNL.

TREE

TREE is a program for the SIV or NRM that provides: ARCHIVE over the network, listing of a directory tree, searching a directory tree for a specified file, deletion of an entire directory tree, wildcard deletion of files from a directory tree, or displaying the total disk space used by a particular user or directory tree. Commands provide for OWNedby, MODIFIED-BEFORE or SINCE controls.

MENU COMPILER

Allows users of the Series IV or NRM to modify the command level menu to include their own commands or to remove commands not often used. Source for the current Series-IV menu line is provided as well as the additional information needed to add Toolbox commands. Menu compiler input is provided in the form of an LL1 parse tree which will require some knowledge of compiler technology to modify.

MSCOPY

MSCOPY is an iNDX utility that allows manipulation of an MS-DOS disk on a Series IV or NRM. Using this program, the Series-IV or NRM can read and write MS-DOS files. Source and generation are provided.

NETWORK CP/M-80

Network CP/M is a package that allows a Series-II or ISIS cluster to run CP/M®-80 and use the NDS-II as a remote file server. A separate license is required for CP/M on each work station. This package is only an interface that allows to use the NDS-II as a file server, the CP/M operating system is not provided. CP/M is available separately as Intel part number SD01CPM80-B-SU. Source is provided for utilities only.

NETWORK CP/M UTILITIES

CP/M	— loads Network CP/M onto the Series II or ISIS cluster.
MAKDSK	— creates a blank Network CP/M disk image.
CDIR	— gives directory of a Network CP/M disk image or CP/M-80 diskette in drive 1 of a Series II.
ADDSYS	— adds CP/M OS to disk image A: created using MAKDSK.
CCOPY	— allows an ISIS user access to CP/M files.
CPMOMF	— converts a program developed under ISIS to a CP/M executable program.
SUCPM	— SUPERUSER facility for CP/M.

BOOTUP

BOOTUP allows an iMDX-580/581 ISIS cluster board to be used in any SBC chassis instead of only a microcomputer development system. BOOTUP is a special monitor PROM which is installed on a standard ISIS cluster board. This board is then installed into any SBC system chassis to provide a diskless work station. The cluster board accesses the NDS-II file system via an iSBC®550 communication controller also installed in the system chassis. Additional ISIS cluster boards may be installed in the same chassis to provide for more users instead of using a Series-II, III, or IV. Up to eight clusters can be used in a single system chassis.

NOTE:
Only object files are provided, the customer must provide his own 2732A PROM. Object files are provided for all formats of Intel PROM programmers.

SERVER

SERVER allows an ISIS Cluster to automatically log-on to the NDS-II network by supplying a username and password from PROM. ISIS will then execute the corresponding initialization file (:f9: ISIS.INI). A useful application of SERVER is to provide additional spooled printer capability to the network by executing PRINCE, another Toolbox application, in an infinite loop from the ISIS initialization file. Some source and generation are provided. All object files are supplied.

PRINCE

PRINCE is an ISIS based spooling program for use with a Series-II, III, IV, or ISIS cluster board in either the stand-alone or networked environments. PRINCE provides support for both parallel and serial printers, including complete XON/XOFF or DTR/DSR printer ready protocols. PRINCE is most effective when used with an ISIS Cluster board and the SERVER PROM. The program features extensive logging capabilities. Source and generation are provided.

PRMSLO

PRMSLO is a PROM image for an ISIS Cluster that sets the default baud rate to 300 or 1200 baud. This enables the cluster board to be used with a modem. Object files only supplied.

UDXCOM.LIB

System library for iNDX specific UDI extensions. This library provides support for MULTIBUS® hardware and software interrupt calls, enable/disable interrupts, read directory expanded, and more. Object code and documentation are provided for this library.

OSXCOM.LIB

System library for internal iNDX operating system extensions. This extensive internal system library provides many system level calls, such as create directory, enable/disable break, change access, change owner, change password, MIP communication calls, and many more. Object code and documentation are provided for this library.

BVOSX.LIB

This library provides operating system support for C language programs in the SMALL model. Normally the programmer would use OSXCOM.LIB and the COMPACT model of compilation. The functions in BVOSX.LIB are the same for the corresponding calls in OSXCOM.LIB, although not all functions are provided. Source and generation are provided.

BVCLIB

BVCLIB is a useful set of C language functions contained in the libraries BVCSLB.LIB and BVCLLB.LIB. BVCSLB.LIB is SMALL model, and BVCLLB.LIB is large model. Functions included are: parse, wmatch, strtok, valid, creat, open, read, write, seek, close, conn_num, str_to_uppercase, str_to_lowercase, plm_to_c_str, c_to_plm_str, err_chk, mark_end. All references to strings are assumed to be C format strings. Source and generation are provided.

SLEEP

This program puts a Series-IV or NRM to SLEEP for the time specified in the (time) parameter. SLEEP can be used in a submit file to execute a program at a certain time. For example, automatically archiving at midnight and then returning to sleep until the next day at midnight and repeating the archive. Source and generation are provided.

ID

ID is an iNDX utility that lists the name of the current user to the current console. It is useful if you forget who you are or need to know who is executing a particular submit file (MAILMAN is a good example of this). Source and generation are provided.

MDS-800 FPORT

INIT800.86 and FPRT are iNDX and ISIS utilities that allow file transfer between an MDS-800 development system and Series-IV over a serial line. Requires S4FPRT.86 (supplied standard with the Series-IV). Source and generation are provided.

DBLIST

DBLIST is an ISIS utility that enhances the operation of the SVCS programming tool set. It can list the entire SVCS database to an output device and may be used to remove deleted variants from a data base directory. Source and generation are provided.

REMOTE Communication with iPDS, Series-II, III, IV

This program gives the remote iPDS, Series-II, III, or IV user complete access to an NDS-II system through an ISIS Cluster board; including file upload and download capability. The program is menu driven and includes: serial channel select, 8253 clock select, break-key select, baud rate select, modem present/not present select, dial/touchtone select, add-to-out call list option. Source and generation are provided.

REMOTE Communication with IBM PC running MS/DOS

This program enables an IBM Personal Computer running MS/DOS to act as a dumb terminal for an ISIS Cluster board connected to an NDS-II network. The ability to upload and download files from the PC to the network is supplied. Source and generation are provided.

REPORT

REPORT is an ISIS utility that reports back on the status of a job that has been EXPORTED to the NDS-II network for execution on a remote job station. The user can add messages to the command file at appropriate positions in the job sequence, and these messages are returned to the ISIS user when encountered. Source and generation are provided.

DIRT

DIRT is an iNDX utility which provides a directory listing with time and date of file creation and modification. Source and generation are provided.

VIEWPASS

VIEWPASS is an iNDX utility provided exclusively for the SUPERUSER. It lists all the usernames on the system, their associated passwords, and their id number. Source and generation are provided.

FDUMP

FDUMP is an iNDX utility that is used to print the contents of a file on the console in one of four possible formats: HEX, BINARY, OCTAL, or DECIMAL. The default is HEX if no option is specified; all formats include a display of the file in ASCII (reverse video on the Series-IV). Source and generation are provided.

CLOCK

CLOCK is a desk clock for use when you have nothing else to worry about. CLOCK displays the current system time on the console of a Series-II, III, IV (iNDX) or ISIS Cluster. Eight and sixteen bit versions are supplied for ISIS and iNDX systems. Source and generation are provided.

IFILES

IFILES is an ISIS-III (N) utility used to identify date/time stamped files in a directory. All of the files that conform to the defined specification will be listed in a savefile. This file can further be used in command files for manipulating the identified files. Source and generation are provided.

LIST

LIST is a utility that copies files to the system printer (:SP: or :LP:). LIST has the following features as enhancements over a normal copy to :SP:.

1) No form feed at the very beginning of a file.
2) Assumes '.LST' for an extension if none is given.
3) Supports multiple copies.
4) Supports multiple files.
5) Supports page breaks.
6) Supports printing of the filename at the beginning of the listing.
7) Converts tabs to spaces if necessary.

Source and generation are provided for ISIS and iNDX versions.

TA

TA is an ISIS based type-ahead utility for the Series-II/III. TA provides a 255 character type-ahead buffer on the Series-II/III. TA requires the iMDX-511 enhanced IOC upgrade, available on most systems manufactured after 1983. Source and generation are provided.

MAILMAN

MAILMAN is an extensive command file that supports multiple network electronic mail when used with more than one NRM and NRM to NRM communications. Source is provided.

CHECKEXIST and CHECKTIME

CHECKEXIST and CHECKTIME are iNDX utilities used to assist the automation of iNDX command files. CHECKEXIST provides a true or false system variable (%status) depending upon the existance of a specified file. A following check of %status within the command file will control the flow of the command file based upon the existance of the specified file. CHECKTIME provides a greater or less than %status by comparing an input time with the system clock for conditional execution of commands in the command file at specified times. Source and generation are provided.

XID

XID, (pronounced "zid") the (X)enix (I)mport (D)aemon, provides XENIX services for any workstation that can access an OpenNET NRM. Thinking of it in another way, XID provides yet another resource for NRM users; a resource much like a spooled line printer or mass storage. In this case, the resource provided is "any job or service that a XENIX box can do; you, as an NDS-II user can gain access to". Source and generation are provided.

REEXPORTER

Reexporter is an iNDX utility that allows OpenNET users (PC's, Xenix, iRMX Systems) to execute batch jobs on NDS-II systems (i.e., VAX/VMS, Model 8001 Series II, III, IV). The utility will execute on a Series-IV, Compilengine or the NRM itself. In brief, it scans special user directories on the NRM looking for command files. If a command file is found, it re"EXPORT"s the command file to a DJC job queue. A log file is generated to allow the OpenNET user to check the success/failure of the job. Source and generation are provided.

XTAR

XTAR is a program that will let you manipulate a XENIX tar diskette at a Series IV. XTAR works only with disks formatted by the /dev/dvf0 device driver on a 286/310 box, or with the /dev/fd048ds96 device driver on a PC/AT. This version will not handle files physically bigger than a single flippy (367104 bytes). Source and generation are provided.

ISIS

The ISIS environment is designed to allow 8080 based Intel tools (such as ASM, PLM LINKER/LOCATOR) to run on an 8086 family system, either iRMX or PCDOS based system. The ISIS environment does not support all ISIS calls, but is sufficient to run 8051 translators and utilities. Hosting ISIS on Xenix-286 systems is possible and installation instructions are included. All object files are supplied.

OAP

OAP is a utility that for security reasons masks the username and password in the PC-Link net use command for increased security. The utility also displays all available servers, by looking at the NETADDR file. Source and generation are provided.

SPECIFICATIONS

Operating Environment

ISIS, iNDX, RMX, XENIX, or PC-DOS operating system. Check description of each tool for specific requirements.

Documentation

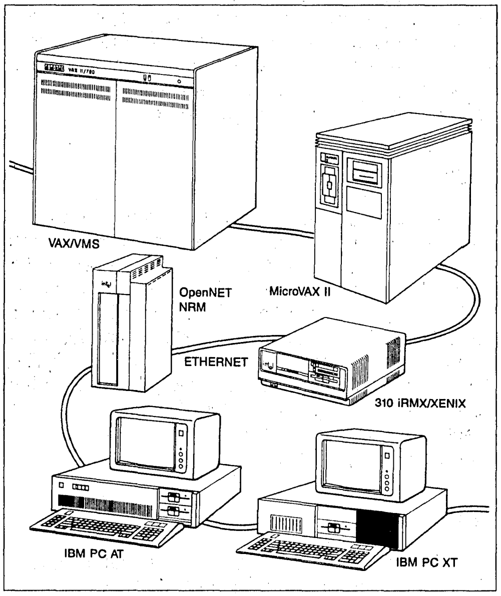
"NDS-II/Series-IV/OpenNET Toolbox"
(122336)

ORDERING INFORMATION

NDS2 TLB	NDS-II/Series-IV/OpenNET Toolbox

VAX/VMS* NETWORKING SOFTWARE
Member of the OpenNET™ Product Family

As a member of Intel's OpenNET™ family of network software, VAX/VMS* Networking software (VMSNET) lets you connect a VAX or MicroVAX II* system to other OpenNET systems. This includes the IBM PC AT, PC XT, Intel's OpenNET NRM (Network Resource Manager), NDS-II NRM (with the OpenNET upgrade kit installed), iRMX®, and XENIX*systems. VMSNET enables a (Micro)VAX system to be configured as a Server System on the OpenNET network, thus allowing any OpenNET Consumer workstation (iRMX, XENIX, MS-DOS) to transparently access files residing at remote (Micro)VAX systems. In addition, VMSNET supports bidirectional file transfer initiated from a (Micro)VAX to all other OpenNET servers.



Product Highlights

- **Connects a VAX and MicroVAXII to the OpenNET Network**
- **Interoperation between VAX/VMS and MS-DOS, iRMX, XENIX, and iNDX systems over a Local Area Network (LAN)**
- **Conforms to the ISO-OSI networking standards**
- **Adheres to ISO 8073 Transport and Ethernet/IEEE 802.3 Standard Communication Protocols**
- **Uses 80186/82586 Processor-based Unibus and Qbus Network Controller Boards**
- **All data stored at the (Micro)VAX is visible to, and can be transparently accessed by, all consumer workstations on the OpenNET network**
- **Enables high speed file transfer/file copy between the (Micro)VAX and OpenNET workstations**
- **Compatible with DECnet***

OpenNET Overview

Intel's OpenNET product family incorporates a set of system and component level LAN products covering all seven layers of the ISO (International Standards Organization) Open Systems Interconnect (OSI) model, and the protocols on which they are based. OpenNET protocols are, whenever possible, established industry standards for each function. Therefore, OpenNET network products can interconnect and interoperate not only with each other, but with the other vendors' ISO-OSI based LANs. An OpenNET network provides a high level of interoperability between heterogenous systems: MS-DOS, VMS, iNDX, XENIX, and iRMX operating system versions are available. Thus, users can tailor their networks to meet their specific needs by incorporating any combination of these diverse systems.

*XENIX is a trademark of Microsoft Corporation. VAX/VMS, MicroVAX II, DECnet are trademarks of Digital Equipment Corp.

ORDER NUMBER 280329-001

Physical Description

The VAX/VMS Networking Software package consists of the appropriate network controller board and the software necessary for the (Micro)VAX to communicate over the OpenNET network. The following sections describe the hardware and software components of VMSNET.

VMSNET Hardware

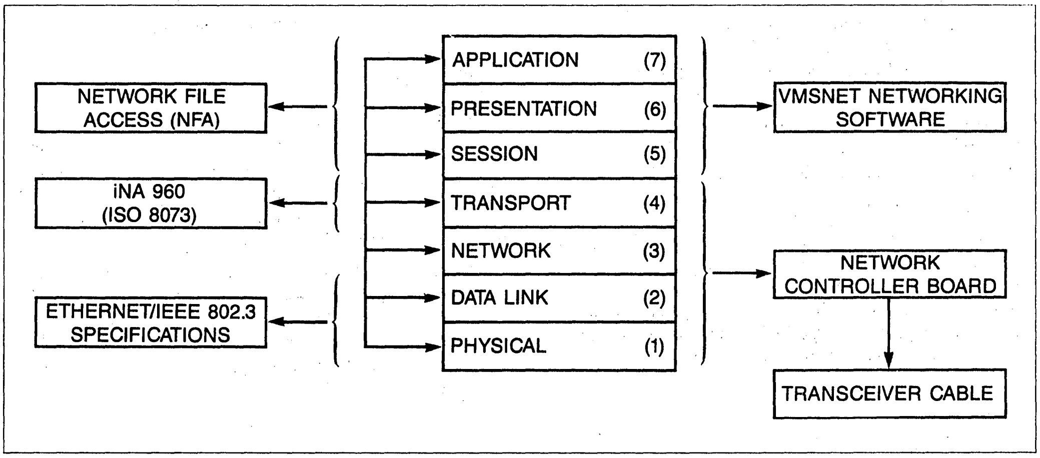
VMSNET comes with one of two types of Ethernet controller boards: a Unibus* board for the high-end VAX or a Qbus board for the MicroVAXII system. Both boards implement the industry standard ISO 8073 transport protocol and Ethernet/IEEE 802.3 physical data link technology. Both boards are high performance, intelligent communications controllers featuring onboard, dedicated Intel 80186/82586 processors which support layers 1 through 4 of the ISO OSI Reference Model. Thus, the Unibus and Qbus* boards perform the CPU tasks associated with lower layer LAN communications protocols, thereby freeing the (Micro)VAX host CPU to concentrate on applications requirements.

Power-up, self-test diagnostics are resident on both the Unibus and Qbus controller. Extended host resident diagnostics are also provided which can be loaded onto the boards to aid in problem resolution. In addition, appropriate internal cables, and chassis mounting hardware are included.

VMSNET Software

The software is supplied on either a 9 track magnetic tape (for high-end VAXs) or on both a TK50 cartridge tape and RX50 5¼-inch disk (for MicroVAXIIs). The following software components are included as part of the VAX/VMS networking software:

- A specially configured version of iNA 960 transport layer software which operates on the network controller boards
- A VMS interface driver which enables VMS programs to access the network controller board
- An implementation of the Network File Access (NFA) protocols (jointly developed by Intel, IBM, and Microsoft) which enables (Micro)VAX users to interoperate with other nodes on the OpenNET network



ISO-OSI VAX/VMS OpenNET Implementation

OpenNET, iRMX are trademarks of Intel Corporation.
* Unibus and Qbus are trademarks of Digital Equipment Corporation.

Functional Description

Transparent File Access

VMSNET provides transparent remote file access capability to the (Micro)VAX through a file server module. The server receives, interprets and executes the command acting as a user to its local file system. Consequently, a PC, iRMX, or XENIX user can work with data files and resources residing at the VAX as if they were resident on his/her system.

File Transfer

VMSNET also provides a set of file transfer utilities that allow (Micro)VAX users with the ability to transfer files that reside on other OpenNET server nodes to the (Micro)VAX or vice-versa. These utilities include copying files, deleting files, listing directories, and a help facility.

DECnet Access

VMSNET will allow consumer access to a file residing on DECnet nodes. The only protocol restriction is that the server will not allow file locking or compatibility mode opens on DECnet file access. The consumer may use logical names to define DECnet pathnames. For example, if "dev" is defined in login.com with an equivalence string of "isodev" user mypassword"::dra 1[user]", the consumer can use "dev" as the first pathname component; the server will automatically use DECnet for the file access:

- net use vms //vms/user mypasswork
- lc //vms/dev
- cp //vms/dev/test.obj/usr/bin

Network Management

A set of network management utilities provide (Micro)VAX users with information and statistics of VMSNET along with the capability to control the execution of the VMSNET server and file transfer utility. To invoke the network utility, the user simply needs to type "NET" in response to the DCL (Digital Command Language) prompt.

Host Requirements

- VAX 750, 780, 782, 785
- VAX 8xxx family
- MicroVAXII
- (Micro)VMS operating system, version 4.2 or later

Physical Characteristics

Software

1. 9 track 1600 bpi magnetic tape
   or
2. TK50 cartridge tape and RX50 5¼-inch disk

Power Requirements

Unibus controller:	+5 vdc (±5%) at 4.5 amps typical, 6 amps maximum −15 vdc (±10%) at .5 amps, 3 amp surge
Qbus controller:	+5 vdc (±5%) at 6 amps typical +12 vdc (±10%) at .5 amps, 3 amp surge

Environmental Characteristics

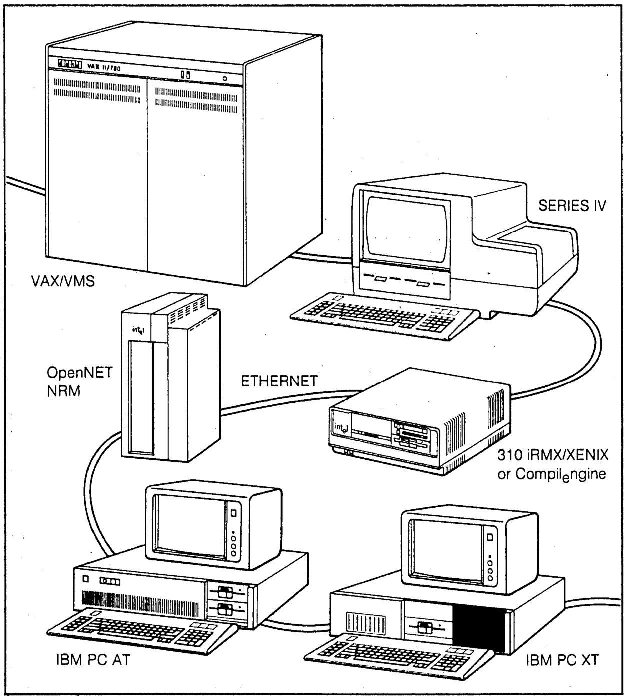
Operating Temperature:	0° to 50°C (32° to 122°F)
Operating Humidity:	Maximum of 90% relative humidity, non-condensing Forced air cooling

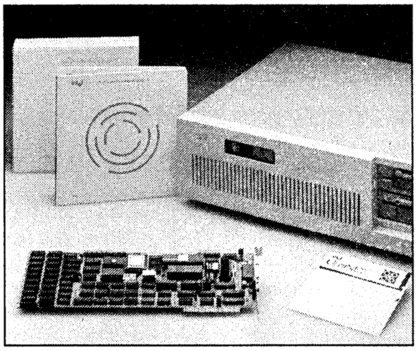
Ordering Information

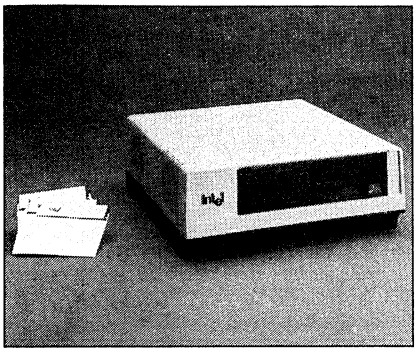
VMSNET	VAX/VMS Networking Software for installation on a high end VAX: consists of a Unibus network controller board with 256KB RAM, a 5 ft. and 10 ft. flat transceiver cables, software on a 9 track 1600 bpi magnetic tape, and an installation and user's guide.
MVMSNET	VAX/VMS Networking Software for installation on a MicroVAXII: consists of a Qbus network controller board with 256KB RAM, an 18 inch flat transceiver cable, software on both TK50 cartridge tape and RX50 5¼ inch disk, and an installation and user's guide.

NETWORKING FOR THE DEVELOPMENT ENVIRONMENT

- **OpenNET™ Network Resource Manager (NRM) provides shared file storage for all workstations**
- **OpenNET PC Link connects personal computers to the network**
- **Compil$_e$ngine offlc ıds compiles from any system on the network**
- **VAX Link for VAX/VMS* network communication**
- **NDS-II NRM OpenNET Upgrade**
- **Ethernet communication speeds**
- **Conforms to industry communication standards (ISO/IEEE)**







ORDER NUMBER 280258-001

The total network development solution based on standards

Intel's open development networking encompasses the needs for existing as well as new Intel OpenNET™ development users. In the lab, Intel protects your investment by allowing you to interconnect existing Intel development workstations and other industry-standard hosts, such as the VAX/VMS* and the IBM PC. This network integrates OpenNET, Intel's open systems strategy for local area networks (LANs). It also ties the development lab, factory and office into a coherent environment.

The OpenNET family implements standards at each level of the International Standards Organization's seven-layer Open Systems Interconnect (OSI) model. For the lab, this includes a range of special networking services to provide your development lab with the power and flexibility needed to solve today's and tomorrow's problems.

A file server tailored to lab requirements

The OpenNET Network Resource Manager file server manages all network workstation requests for central resources, including file access, print spooling, tape back-up, remote job execution queue management, program management and network maintenance functions. The NRM, unlike many office file servers, features a full-featured, protected, hierarchial file system. The NRM supports transparent access to this file system from any OpenNET consumer (e.g., MS-NET, XENIX*, iRMX™ 86) as well as from Intel's Intellec Model 800, Series II, III and IV Systems.

Two OpenNET models are available: the MAXI, with a 140MB Winchester and a 60MB tape storage, and the MINI, with a 40MB Winchester only. Both are 8 MHz, 80286-based super-microcomputers with 1MB of zero wait-state RAM. And, both are optimized for file access using techniques such as caching of most recently used tracks, very fast disks, fast disk-seeking algorithms and communications boards with their own dedicated microprocessors.

Program Management Tools (PMTs) decrease the time spent tracking program/module changes and manually generating programs, giving software engineers more time for design, development and testing.

A remote job execution facility provides automatic network load balancing.

Transform your PC from an individual workstation to team member

The OpenNET PC Link enables users to connect their IBM PC XT/AT or compatibles to the OpenNET Network and to transparently access and share files and printers on an OpenNET NRM, NDS-II NRM, iRMX and XENIX-based file servers. OpenNET PC Link features an 80186/82586 microprocessor-based Ethernet/IEEE 802.3 expansion board, Microsoft networking software (MS-NET) and iNA960 transport software (ISO8073-compatible).

Compil$_e$ngine: a shared network resource

Compil$_e$ngine is a shared, networked system optimized to offload compile and link/locate jobs from any workstation or VAX on the network. It is an 80286-based supermicro-computer that compiles faster than any workstation and requires no terminal to operate. Moreover, because it supports two partitions, it can be used as a software work-station at the same time it is being used as a shared resource.

This product can be connected to an NDS-II NRM or OpenNET NRM.

Workstation requirements

WORKSTATION	SOFTWARE REQUIREMENTS	HARDWARE REQUIREMENTS
NDS-II workstations		
Compil$_e$ngine	—	—
Model 800	—	PIMDX 455
Series II/III	—	PIMDX 455
Series IV	—	PIMDX 456
Cluster Chassis	NDS2TLB	PIMDX 455
VAX	IMDX 392; [VMS V4.2]	[DEUNA* Board]
OpenNET workstations		
PC DOS	DOS V3.1	PCLNK
System 310 XENIX	XNX-NET R1.0	iSXM 552
System 310 RMX 86	RMX-NET R1.0	iSXM 552

[] Available from DEC
NOTE: Interconnecting hardware (cables, transceivers) requirements are not included in the above chart.

Ordering Information

iMDX 460-140T	OpenNET NRM (MAXI model).
iMDX 460-40	OpenNET NRM (MINI model).
iMDX 555	NDS-II NRM OpenNET Upgrade Kit.
iMDX 485CE	Compil$_e$ngine.
NDS2TLB	Network Software Toolbox.
iSYP 312	Floor stand which encloses either the OpenNET NRM or the SYS 311 peripheral expansion box.

NDS-II/VAX Link

This Ethernet-based link between an OpenNET NRM or an NDS-II NRM and a DEC* VAX/VMS microcomputer allows VAX users to copy files from the VAX to the NRM for debugging, in-circuit emulation and testing. With the remote job execution feature, VAX users can send CPU-intensive jobs to idle workstations (such as the Compil$_e$ngine) for execution. Conversely, NRM users can send jobs for remote execution on the VAX.

NDS-II NRM OpenNET™ Upgrade

This product allows your NDS-II NRM to double as an OpenNET file server for PC, XENIX and iRMX workstations; files on the NRM may be transparently accessed by any workstation on the network.

Peripheral Expansion Option

The OpenNET NRM supports 40 or 140MB of Winchester storage on a single disk drive. Mass storage can be expanded to 460MB on the MINI NRM and 560MB on the MAXI NRM, using the 311 peripheral system.

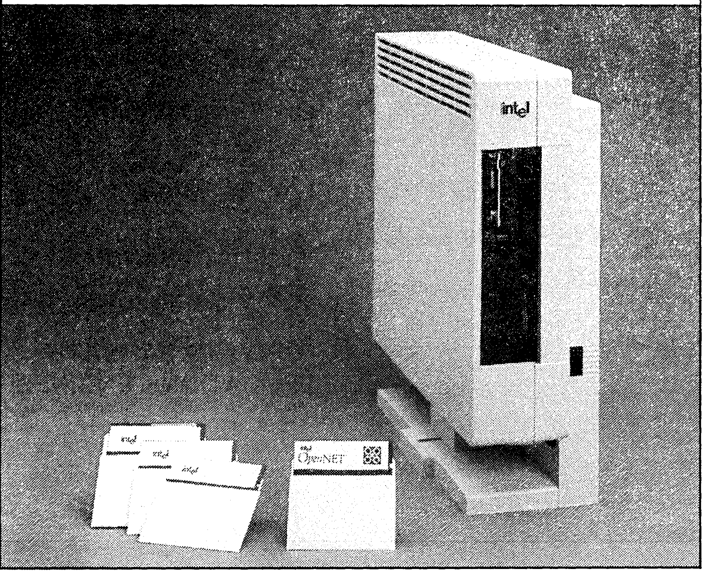
*VAX/VMS, DEC & DEUNA are trademarks of Digital Equipment Corp.
XENIX is a trademark of Microsoft Corp.

Workstation Kits

PIMDX 455	NDS-II Workstation Upgrade Kit for any Series II/85, Series III, or Model 800 to connect to the OpenNET NRM or NDS-II NRM.
PIMDX 456	NDS-II Workstation Upgrade Kit for the Series IV.
PIMDX 581	ISIS Cluster Board Package.
IMDX 392	VAX Link R2.1 for VAX/VMS connection to the NRM.
PCLNK	OpenNET PC Link hardware and software kit to connect the PC XT, PC AT, and compatible systems to the NRM via the OpenNET network; requires DOS 3.1 or higher.
RMXNETKITWRI	iRMX Networking Software for a 286/310 system running the iRMX 86 operating system to connect to the NRM via the OpenNET network.
SXM 552S	Ethernet-based Single Board Network Communication Engine for 310 systems.
XNXNETNRIKIT	OpenNET-XenixNET Networking Kit. Includes iNA 961, SXM 552 and XenixNET pass-through networking software.

Interconnecting Hardware

PIMDX 457/458	Transceiver cables (10/50 meters) (two are required for an OpenNET NRM; one is required for a Compil$_e$ngine).
PMDX 3015	Transceiver for Ethernet coaxial cables (at least two are required unless an Intellink is used).
iDCM 911-1	Intellink module (the OpenNET NRM uses two ports).
PIMDX 3016-1/ 3016-2	Ethernet coaxial cable (25/50 meters).



intel®

APPLICATION NOTE

AP-240

October 1986

Using Archive To Efficiently Control a Network

SRIVATS SAMPATH
DSO APPLICATIONS ENGINEERING

Order Number: 231476-001

INTRODUCTION

The onset of large scale software projects has generated additional needs in all levels of the development environment. The need for a sophisticated source and version control system and efficient disk management becomes particularly important as the project team grows. This need is more pronounced at the software management level, where operating the project on schedule is of prime importance. Efficient disk management includes keeping the disk free of redundant files and keeping copies of older versions somewhere other than the disk itself. In other words "ARCHIVING" all previous versions onto another storage media that is inexpensive, reliable and transportable is key. One storage media that meets all these requirements is the NDS II tape sub-system which forms an integral part of the development environment. Moreover, the actual archive process should be easy to use, preferably automatic and should not be a drain on resources. The ability to manage mass storage devices efficiently translates into a substantial increase in productivity for everyone. Intel realizes this need and has developed a solution that is tailored towards helping the NDS-II system manager efficiently control the development project. We introduced the TAPE SUB-SYSTEM on our Network to provide an inexpensive, reliable and transportable media, and a utility called ARCHIVE to make actual disk management both user friendly and automatic.

ARCHIVE

The ARCHIVE utility performs file backup and restoration by copying files and directories to magnetic tape or other secondary storage devices. This utility is executed at an NRM console, and with its powerful set of options, it positions itself as an invaluable tool for efficient disk management.

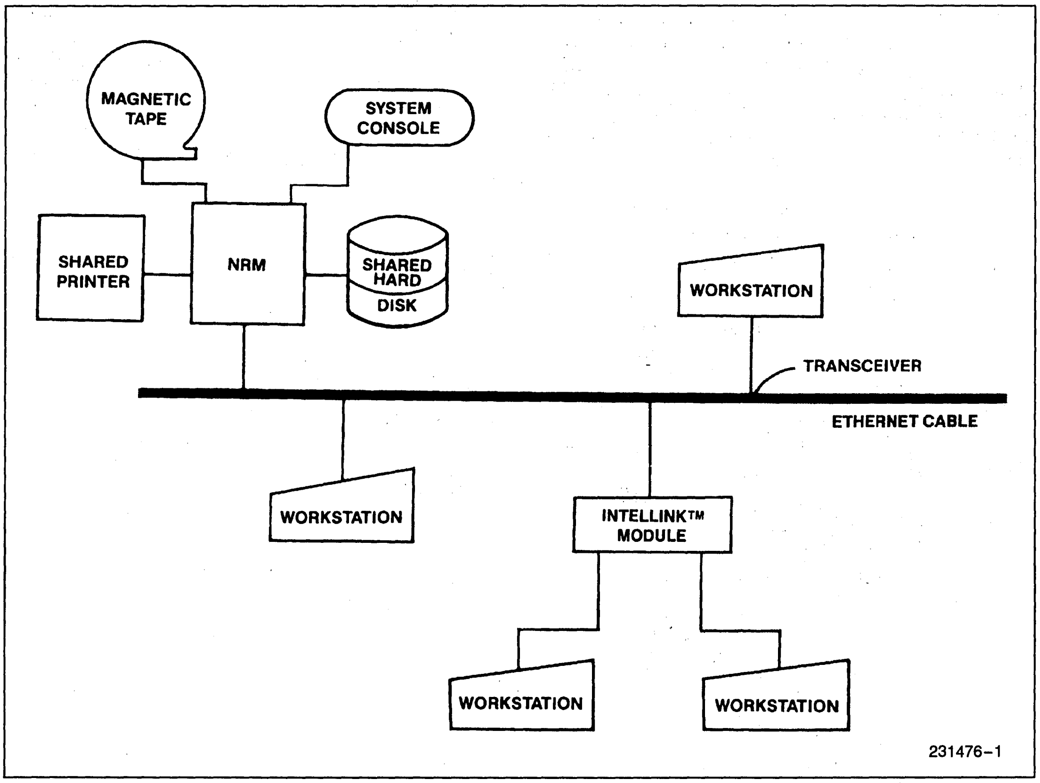


Figure 1. The Network and Its Components

WHY USE A TAPE?

Magnetic tape is regarded as the most useful storage device in the computer industry. In spite of its sequential structure, tape answers a number of requirements that can not be met by any other conventional mass storage devices. The most strong argument in favor of tape is its portability - the ability to transport tape with the minimum overhead and damage during transit makes it an extremely attractive media. Moreover, storing tapes is much more organized and efficient than storing diskettes. All these arguments lead to a single conclusion: "The Magnetic Tape should form an integeral part of any development environment". The ARCHIVE utility and the tape drive together form the foundation for effective disk management, bringing about a more productive environment for any development project.

Additionally, the tape is a safety net for one of those rare disk crashes. Having backups on tape will reduce the amount of data loss in the event of a fatal disk crash. Additionally, completed projects can be saved on ape to provide more disk work space. Having these projects on tape minimizes the effort in reloading all the data if major bugs are found, or if an update is involved. Multi-project sites can benefit from the fact that tape allows easy transporting of data.

Tape backup can be classified into incremental backup and tape streamer. Tape streamer allows volume copies, with every record on the disk copied onto tape. It is a mass data transfer from one storage device to another. This brings about a lot of overheads when only some parts of the disk need to be backed up or restored.

Incremental tape backup treats the tape as a random access device similar to a disk. Files can be added, appended or deleted, just as in a disk. This feature allows selective backup onto tape; thus eliminating the need for a mass copy operation when only a few files need to be archived or restored.

The tape drive on the NRM is an incremental backup device and ARCHIVE has been designed to use this to the fullest extent.

HOW DOES ARCHIVE HELP?

ARCHIVE has been designed to let the NDS II system manager operate the development project at maximum efficency. Using its various options, the system manager can selectively archive files and directories onto tape, employing various qualifiers such as date accessed, created, modified, before, since, on, etc.. These qualifiers will be discussed in a later chapter (Invocation and Syntax) with examples. The options are essential for NDS-II system managers to perform selective archives of files and directories.

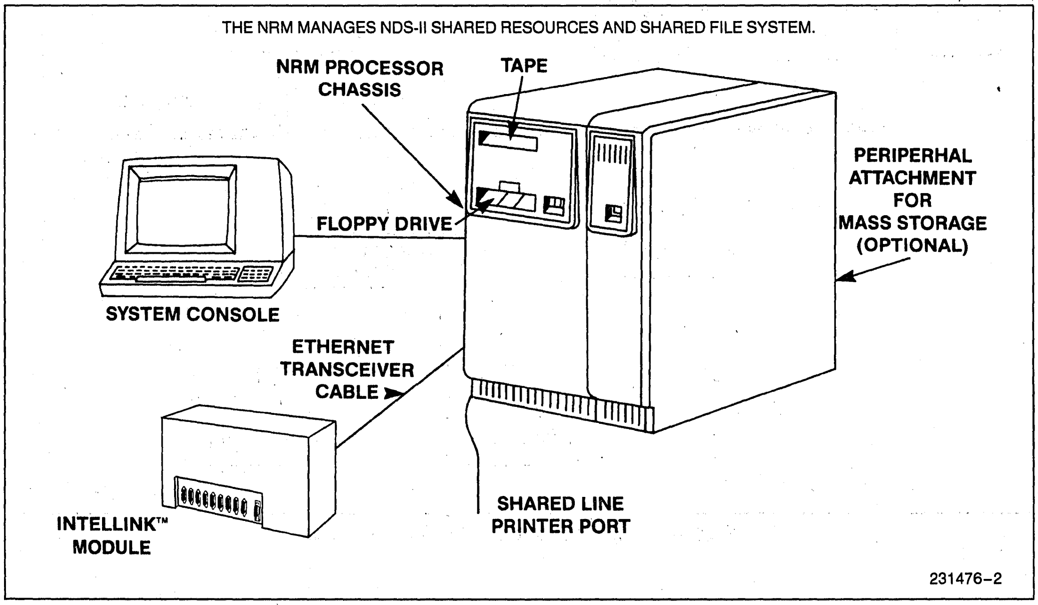

THE NRM MANAGES NDS-II SHARED RESOURCES AND SHARED FILE SYSTEM.
NRM PROCESSOR CHASSIS
TAPE
PERIPERHAL ATTACHMENT FOR MASS STORAGE (OPTIONAL)
FLOPPY DRIVE
SYSTEM CONSOLE
ETHERNET TRANSCEIVER CABLE
INTELLINK™ MODULE
SHARED LINE PRINTER PORT
231476-2

Figure 2. Network Resource Manager (NRM)

ARCHIVE can also be used in a submit or a batch file. This saves the NDS II system manager from having to type in the whole command syntax every time an ARCHIVE must be performed adding another step toward improved productivity.

Utilities like SLEEP, wakes up the system at a specified date or time, goes a long way in automating ARCHIVE. A submit file is invoked at the NRM that wakes up the system at a specified time (preferably near midnight when system load is low), archives all qualified files onto tape, then 'goes back to sleep' again. This is an important factor eliminating the need for operator intervention at any time and automating the entire process.

ARCHIVE frequency depends on the particular application and system load. It is recommended that ARCHIVE be performed at least once a week. However, in large project implementations (i.e 6 or more design engineers involved in generating or modifying more than 100K of code), ARCHIVE should be performed automatically each night. This ensures that even if a disk crash occurs, data loss is restricted to a single day's work.

All the features incorporated in ARCHIVE make it an attractive solution for effective version control and disk management. It is a productivity tool that no system manager should do without.

DATA LAYOUT ON TAPE

Tape is a sequential structured media, with all files and directories sequentially stored, but it maintains the hierarchic file structure of a disk. Every time an ARCHIVE is performed, a LOGICAL VOLUME is created, containing any number of files, from NULL to a set of files residing on a device. Each LOGICAL VOLUME has a VOLUME NUMBER and a HEADER associated with it. VOLUME NUMBERS start with 1 upwards. The HEADER is the source path name. For example:

```
ARCHIVE /WDO/USERS.DIR TO CTO
```

creates the first record onto tape and gives it the VOLUME NUMBER 1 and HEADER /WD0/USERS.DIR. Files and sub-directories under USERS.DIR will be copied in a TOP DOWN order. The same rules apply to all the subsequent sub-directories. The data layout on tape is shown in Figure 3.

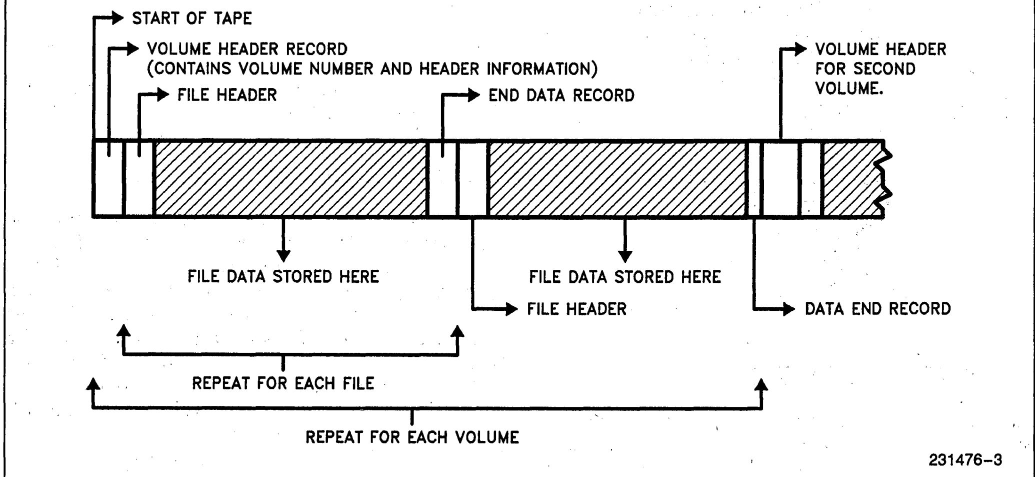


Figure 3. Format of Data on the Tape

INVOCATION AND SYNTAX

ARCHIVE, with its powerful set of options, gives the user flexibility in effectively managing the disk. The syntax of ARCHIVE is given below. During operation at the NRM, the system syntax builder prompts the user for options so none of the options have to be memorized.

The ARCHIVE syntax consists of a set of qualifiers and a set of switches. QUALIFIERS are options that qualify a file or directory for copying, allowing the user to selectively choose files and directories for archiving. SWITCHES are sets of controls that enable the user to actually control the I/O operation.

SYNTAX:

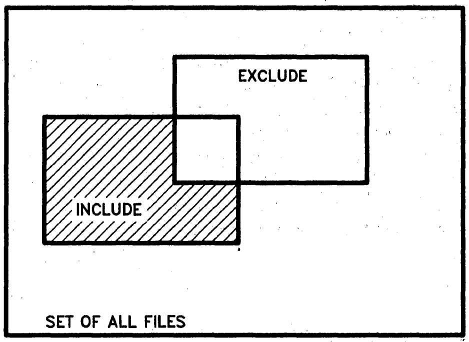
```
 ARCHIVE source TO destination
   < O P T I O N S >

OPTIONS ARE:
1. QUALIFIERS:
INCLUDE,    EXCLUDE (files that were . . . )
              ACCESSED/CREATED/MODIFIED
                  BEFORE / SINCE / ON
                     TODAY / date
                         hour

              DIRECTORY (directory name, . . . )
                OWNEDBY (Owner name, . . . )
                   FILE (path-name, . . . )
                       AND/OR . . .
2. SWITCHES: APPEND
             DELETE
             ERASE
             LOG log-file-name
             NAME physical volume name
             NOUPDATE
             QUERY
             UPDATE
             VOLUME (logical volume number )
```

ARCHIVE QUALIFIERS

Qualifiers enable the user restrict the number of files to be archived. and discriminate against any file by time stamps (time created, modified, etc), the owner, file names and even by parent directory. These qualifiers have no limit to their length or order of appearance, an may be specified using the keywords INCLUDE/EXCLUDE.

Include/Exclude

INCLUDE specifies the files that are to be included in the command, while EXCLUDE lists the file that can not be archived if the qualifying condition is met. EXCLUDE has precedence over INCLUDE; therefore, when both keys are used (INCLUDE files, EXCLUDE files) the following set of files will be archived:



231476-4

/ : indicates files included

The following is the list of all acceptable keywords for INCLUDE/EXCLUDE:

1. ACCESSED/CREATED/MODIFIED

These switches compare the time specified in the time qualifier to the last time the file was accessed, or the time it was created, or the time it was last modified. If this time agrees with the time qualifier condition, then the file is qualified. The time qualifier is required for ACCESSED and CREATED and is optional for MODIFIED. If the time is not specified with the MODIFIED switch, a default value of SINCE LAST-ARCHIVE-DATE will be used. This default value will qualify all the files which:

a. Were modified since last ARCHIVE.
b. Were created since last ARCHIVE.
c. Were created or modified prior to last ARCHIVE but, through use of qualifiers, they were somehow EXCLUDED from being archived earlier.

Time Qualifiers:

The time qualifiers allow the user to specify an instant in time which is used in a comparison with the time a file was last accessed, created, or last modified to qualify the file for archiving:

— BEFORE Allows specifying a file accessed, created, or modified BEFORE a specific date.

— SINCE Allows specifying a file accessed, created, or modified SINCE a specific date.

— ON Allows specifying a file accessed, created, or modified ON a specific date within a 24 hour period.

— date/TODAY

For neither BEFORE, SINCE, and ON a does default date value exist. Date could be specified in two forms, either by using TODAY switch, which would read the current date from the system, or by actually specifying the date in the form of mm/dd/yy. An optional time of the day (in hours) in hh:mm:ss form with a default value of 00:00:00 can be used.

— hour

Time of the day can be specified in hh or hh:mm or hh:mm:ss. The hour qualifier is to be in parenthesis.

Examples:

```
ARCHIVE /WDO TO CTO INCLUDE CREATED BEFORE 12/21/84
     ;would archive all files created before DEC 21.
     ;a time default of 00:00:00 would be used.
ARCHIVE /WDO TO CTO EXCLUDE MODIFIED SINCE 10/10/83 ( 10:11:22 )
     ;archives all files, exclude those which were modified
     ;since 10:11:22 on October 10th, 1983.
ARCHIVE /WDO TO CTO INCLUDE ACCESSED ON TODAY , EXCLUDE & CREATED BEFORE
      10/26/83 AND MODIFIED SINCE 10/24/83 ( 10:11:12 )
     ;archives all thefiles which were accessed today, and
     ;exclude those which were created before October 26th
     ;and were somehow modified since 11 minutes and 12 seconds
     ;after 10 AM on October 24th.
ARCHIVE CTO TO /WD1/DIR1 INCLUDE ACCESSED ON TODAY , EXCLUDE & CREATED
 BEFORE 10/24/83 AND MODIFIED SINCE 10/26/83 ( 10:11:12 )
     ;all files on the tape which were last accessed today
     ;( exclude archive access itself ) will copied to the
     ;/WD1/DIR1 directory. Files which were CREATED before
     ;October 24,83 and were MODIFIED since October 26, 83
     ;will be excluded.
```

2. DIRECTORY/FILES/OWNEDBY

Allows the user to specify qualifiers other than time such as owner of files, directories, etc.

— DIRECTORY Allows the user to specify particular directories to be included or excluded in ARCHIVE. The directory could either be the full path name of the directory or partial name from where source-name left off. DIRECTORY does not accept wildcard characters. However, logical names are allowed.

— FILE Allows the user to specify particular files to be included or excluded in archive. The file name, in order to be recognized, should only be the filename, not a path name. Wildcard characters are accepted.

— OWNEDBY Allows archive select files on the basis of owner's name.

3. AND/OR

AND/OR allows the extension of the qualifying conditions within a qualifying set. AND/OR can not be intermixed within a qualifier set defined by one INCLUDE or EXCLUDE.

4. COMMA

Comma is the separator (delimiter) between INCLUDE and EXCLUDE. In English, it makes sense to use AND in between INCLUDE and EXCLUDE. However, in ARCHIVE, you can not use anything other than Comma to separate INCLUDE/EXCLUDE. Example:

```
ARCHIVE /WDO TO CTO INCLUDE ACCESSED
ON TODAY , EXCLUDE & CREATED BEFORE
10/26/83 AND MODIFIED SINCE 10/24/83
( 10:11:12 )
```

ARCHIVE SWITCHES

SWITCHES are controls that ARCHIVE gives the user to selectively copy files and directories to/from tape or any other storage device. We will discuss each of these switches in depth to highlight the versatality of ARCHIVE.

1.0 Append, Volume

Every time ARCHIVE is issued onto tape, a Logical Volume is created. This logical volume can consist of one file or as many as all files residing on a particular device. Unless otherwise specified ARCHIVE always starts from the beginning of a tape. The tape is rewound and the header information is written followed by the actual copy operation which copies all qualified files from the beginning of tape. Using the Append switch allows the user to have more than one volume or a related group of files on a single tape .Now instead of starting from the beginning of a tape ARCHIVE searches through the tape for the volume name specified. Default for Append is the last volume on tape.

ARCHIVE will always overwrite an existing volume if Append switch is not specified. Append to an empty tape is not valid as ARCHIVE will not know what to Append the new record to.

Recommendation: The tape should be dismounted only after ARCHIVE signs back on with the message 'ARCHIVE COMPLETE'. Use the ERASE option when writing to a new tape.

Examples: (WITH A NEW TAPE)

```
ARCHIVE   /WINIO/USERS.DIR TO TAPEO APPEND
                 ; This is an ERROR. No previous volume on tape
                 ; to append new volume to
ARCHIVE   /WINIO/USERS.DIR TO TAPEO
                 ; This will create header for Volume #1 and then
                 ; copies all the USERS.DIR files and sub-directories
                 ; to the tape.
ARCHIVE   /WINIO/SYSTEM.DIR TO TAPEO APPEND
                 ; This creates a new volume on the tape (Volume # 2)
                 ; and adds all SYSTEM.DIR files and sub-directories
                 ; to the tape at Volume #2
ARCHIVE /WINIO/ISIS.SYS/FILES TO TAPEO APPEND VOLUME 3
                 ; This skips to the third volume on tape ,writes
                 ; the header for Volume #3 and then copies all
                 ; files and sub-directories to Volume #3
```

2.0 Delete

This switch instructs ARCHIVE to delete all qualified files on the disk after they have been copied onto tape or disk. It is very useful when backups of older versions are performed. Once the archive process has been completed, all the old files are deleted from the source disk giving the user a better control over managing disk files. This is a disk only option.

Recommendation:

It is recommended that the user archive to tape first, using a LOG option and ascertaining that the files exist on tape. Then, he repeats the ARCHIVE to :BB: with the delete switch on to delete all the qualified files from disk. This will eliminate any possibility of deleting files without first archiving them.

3.0 Erase

This option causes the tape to be erased before any write operation is performed. ERASE goes over all tracks on the tape and erases everything written on it. ERASE and APPEND cannot be used simultaneously, since one erases the tape and the other tries to append to non-existent volumes. ERASE is a tape otion.

Recommendation:

Use the ERASE switch when archiving onto tape the first time. Use Append for subsequent logical volumes.

4.0 Log file—name

The LOG option will redirect all console messages to a specified LOG file. Errors generated because of LOG file existence will not abort ARCHIVE. (i.e. if a log file already exists it will be automatically overwritten by ARCHIVE)

Recommendation:

It is good practice to redirect console output to a LOG file when a sufficiently large ARCHIVE is being performed, keep a record of all succesful archives. This LOG file should be listed and stored along with the tape.

5.0 Name physical_volume_name

The first time an ARCHIVE is issued to a tape, using the NAME option will associate the physical_volume_name with that tape. This option ensures that the right tape is used when reading from or writing to the tape. When the NAME switch is specified the name on tape will be compared against the name on the ARCHIVE command line. ARCHIVE will not continue if names do not match. The default physical_volume_name is ARCHIVE, meaning that if a NAME option was not specified during the first write operation to tape, it will be named ARCHIVE.

Recommendation:

The use of logical sounding names for the physical—volume—name of the tape is good practice. This helps in fast identification of the tape being used. Names like PROJECT1 and VERSION1.0 are good names while THIS.IS.IT and LATEST are not. The physical—volume—name should not be more than 14 characters long.

6.0 Noupdate

When archiving information from any source to a hard disk, if an existing file is encountered, NOUPDATE instructs ARCHIVE not to copy over the existing files. Thus if a file is on the disk and there is a matching file name on that tape, archive from the tape to the disk will not update the contents of the file when the NOUPDATE switch is used. This option is a default switch in submit files. If neither UPDATE nor NOUPDATE is used, the user will be queried whether the existing file should be deleted.

Recommendation:

The specified default for ARCHIVE in a submit file is NOUPDATE. However, the default for ARCHIVE in a submit file is similar to the QUERY command. If files being restored already exist, ARCHIVE will prompt the user for deletion. It is recommended that UPDATE or NOUPDATE option be specified within a submit file.

7.0 Query

This causes ARCHIVE to prompt the user for every data and directory file in the source directory, then waits for confirmation. When the user is prompted regarding a directory file, and the user chooses not to copy that directory file, none of the files and sub-directories in that directory can be archived. The default is no QUERY. QUERY used in conjuction with NOUPDATE prompts the user for every qualified file in the source directory. When confirmed that the file exists in the destination directory, the user will be informed that the file exists at the destination directory, but the file will not be copied over. QUERY used in conjunction with UPDATE prompts the user for each file in the source directory. Once confirmed, it will copy the qualified files to the destination regardless of their existence.

```
ARCHIVE   /WDO/USERS.DIR TO CTO NAME TAPE1
                ;archives every file and directory in USERS.DIR
                ;onto the tape. The tape will be named TAPE1
                ;from now on. If an attempt is made to access
                ;or write more files onto the tape with the
                ;NAME switch on, TAPE1 is the only name that will
                ;be accepted by ARCHIVE.
ARCHIVE   /WDO/USERS.DIR/MINE.DIR TO CTO NAME TAPE1 APPEND
                ;would append MINE.DIR to the tape.
ARCHIVE   /WDO/USERS.DIR/YOURS.DIR TO CTO APPEND
                ;would still work fine and appends the
                ;new directory YOURS.DIR to the tape.
ARCHIVE   /WDO/USERS.DIR/WHOSE.DIR TO CTO NAME TAPEO
                ;would be rejected with the message:
                ;RIGHT VOLUME EXPECTED.......
```

8.0 Update

This switch is the exact opposite of the NOUPDATE switch. If UPDATE is specified, all the qualified files on the tape will be copied to the destination directory, despite the previously existing files in the destination directory.

Example:

Assume that files F1 and F2 are in /W1/D1 and directory file D2 is in /W1. Also assume F2 exists at /W2/D1.

```
> ARCHIVE /W1 TO /W2 QUERY <CR>
  iNDX-N11 (V2.8) ARCHIVE, V2.8
  10/26/84 11:12:33 DIRECTORY = /W1
  COPY /W1/D1 TO /W2/D1 ? Y <CR>

  DIRECTORY = /W1/D1
  COPY /W1/D1/F1 TO /W2/D1/F1 ? Y ©
  COPIED /W1/D1/F1 TO /W2/D1/F1
  COPY /W1/D1/F2 TO /W2/D1/F2 ? Y <CR>
  File Already Exists
  Pathname = /W2/D1/F2
  Delete Existing File ? Y <CR>
  COPIED /W1/D1/F2 TO /W2/D1/F2
  COPY /W1/D2 TO /W2/D2 ? N <CR>

  ARCHIVE COMPLETE

> ARCHIVE /W1 TO /W2 QUERY NOUPDATE ©
  iNDX-N11 (V2.8) ARCHIVE, V2.8
  10/26/84 11:12:33

  DIRECTORY = /W1
  COPY /W1/D1 TO /W2/D1 ? Y <CR>
  DIRECTORY = /W1/D1
  COPY /W1/D1/F1 TO /W2/D1/F1 ? Y <CR>
  COPIED /W1/D1/F1 TO /W2/D1/F1
  COPY /W1/D1/F2 TO /W2/D1/F2 ? Y <CR>
  File Already Exists Pathname
  = /W2/D1/F2
  COPY /W1/D2 TO /W2/D2 ? N <CR>

  ARCHIVE COMPLETE

> ARCHIVE /W1 TO /W2 QUERY UPDATE <CR>
  iNDX-N11 (V2.8) ARCHIVE, V2.8
  10/26/84 11:12:33
  DIRECTORY = /W1
  COPY /W1/D1 TO /W2/D1 ? Y <CR>
  DIRECTORY = /W1/D1
  COPY /W1/D1/F1 TO /W2/D1/F1 ? Y <CR>
  COPIED /W1/D1/F1 TO /W2/D1/F1
  COPY /W1/D1/F2 TO /W2/D1/F2 ? Y <CR>
  COPIED /W1/D1/F2 TO /W2/D1/F2
  COPY /W1/D2 TO /W2/D2 ? N <CR>

  ARCHIVE COMPLETE
```

9.0 Volume

The first time an ARCHIVE command is issued, one logical volume will be created on the tape. Subsequent ARCHIVE's to the tape using the APPEND switch create additional logical volumes on the tape. For instance, one ARCHIVE without APPEND and three more ARCHIVE's with APPEND create four logical volumes on the tape. If the user is restoring information from the tape, not specifying volume number restores all the logical volumes on the tape. Specifying a non-existent number causes an error message and aborts the command. A valid volume number searches that particular logical volume for the qualified files and restores only files from that specific logical volume.

Example:

```
ARCHIVE   /WDO/USERS.DIR TO CTO
               ;erases the tape and copies USERS.DIR to
               ;the tape as logical volume 1.
then
ARCHIVE   /WDO/MISC.DIR TO CTO APPEND VOLUME 3
               ;causes an error message, because logical
               ;volume number 2 is not created yet.
then
ARCHIVE   /WDO/MISC.DIR TO CTO APPEND or
ARCHIVE /WDO/MISC.DIR TO CTO APPEND VOLUME 2
               ;creates the second logical volume and
               ;copies all the files from MISC.DIR to it.
then
ARCHIVE   CTO TO /WDO/DIR1 VOLUME 3
               ;generates an error message because
               ;logical volume 3 does not exist.
then
ARCHIVE   CTO TO /WDO/DIR1 VOLUME 2
               ;copies to DIR1 all the data and directory
               ;files which were under /WDO/MISC.DIR and
               ;were archived to the tape. In a sense, the
               ;subtree starting from /WDO/MISC.DIR will
               ;be added to DIR1.
```

DATA RESTORATION FROM TAPE

ARCHIVE allows data restoration from tape onto disk, facilitating easy recovery from a disk crash without a significant loss of data. Reopening a project simply involves reloading all data archived onto tape. This also simplifies multi-site projects, where data can be transported and reloaded from one site to another.

In order to restore data from a tape, the user can use the device name CT0 and restore all information from tape to disk. Or the user can specify a pathname to a directory on tape and restore only that directory and associate files and sub-directories. Finally one can specify a particular VOLUME and restore information stored under that volume. Examples:

Assuming that /WD0/USERS.DIR/TEMP.DIR is empty and the tape has three records (i.e. LOGICAL VOLUMES)

APPEND and ERASE are switches that can be used only when archiving onto tape. DELETE, NOUPDATE and UPDATE switches can only be used with a disk.

```
Record ▪1 (Volume Number 1)
Header /WD1/USERS.DIR/TEMP1.DIR
FILES and DIRECTORIES
     /WD1/USERS.DIR/TEMP1.DIR/FILE1
     /WD1/USERS.DIR/TEMP1.DIR/FILE2
     /WD1/USERS.DIR/TEMP1.DIR/FILE.DIR/FILE3
     /WD1/USERS.DIR/TEMP1.DIR/FILE.DIR/FILE4
     /WD1/USERS.DIR/TEMP1.DIR/FILE.DIR/PASCAL.DIR/FILE5
RECORD ▪2 (Volume Number 2)
Header /WD0/MISC.DIR/TEMP2.DIR/TEMP3.DIR/ F1
FILES and DIRECTORIES
     /WD0/MISC.DIR/TEMP2.DIR/TEMP3.DIR/FILE1
     /WD0/MISC.DIR/TEMP2.DIR/TEMP3.DIR/FILE2
-- ARCHIVE CT0 TO /WD0/USERS.DIR/TEMP.DIR
```

would copy all files from tape onto disk.

```
-- ARCHIVE CT0 TO /WD0/USERS.DIR/TEMP.DIR VOLUME 2
```

would copy all files in VOLUME 2 to disk.

APPENDIX A

SLEEP:

SLEEP is a utility, available in the Network/Series IV toolbox, that executes at an NRM or SERIES IV, allowing the user to delay the execution of certain programs until a certain time. This can be included in a submit file and made to execute continuously. The sample/submit file looks like this:

```
Repeat

Sleep til 23:30:00

ARCHIVE /WDO/USERS.DIR TO CTO INCLUDE ACCESSED ON TODAY APPEND

End
```

This submit file will run forever at the NRM console and will wake up at midnight do all the archives, then go back to sleep again. Since sleep runs on the foreground at the NRM, a Cntr-C has to be performed if the user must utilize the NRM terminal for some other purpose.

This is a very useful utility in conjunction with ARCHIVE as it makes the whole process automatic and eliminates the need for operator intervention.

ACKNOWLEDGMENTS:

I would like to take this opportunity and thank Bahram Saghari in DSO Software Support for his contribution towards this Application Note. All examples on ARCHIVE were supplied by his group.

intel®

APPLICATION NOTE

AP-242

October 1985

Additional Printer Support for the NDS-II System

CHRIS FEETHAM
DSO APPLICATIONS ENGINEERING

Order Number: 231478-001

INTRODUCTION

Using printers for hard copy of data has long been necessary in most computer systems. Software engineers use printers primarily for software program listings, but increasingly, letter quality printers are being used to generate memos, reports, and other business documents, rather than queueing them up at the secretary's typewriter. Additionally, with the cost of computer terminals and network connections declining, it is becoming rare for the business professional not to have immediate or direct access to a terminal with some type of word processor available. The ability to send hard copy directly to a printer rather than waiting for a typist to re-type the input is a productive benefit for everyone.

THE NDS-II NETWORK

With Intel's advanced Network Development System II (NDS-II), development systems are connected into a network using Ethernet. Additionally, each development system has the ability to host several ISIS Clusters that use low cost serial lines to support the terminals. The complete product line is described in the NDS-II System description (refer to Appendix D for complete details).

With low cost terminals available to everyone, including engineers, managers, and secretaries, files and data can be shared and manipulated directly on the network, reducing the many intermediate steps required in producing a final document. The addition of CPM/80 coupled with the industry standard Wordstar word processing package, available for the NDS-II system (refer to Appendix D for details), further increases secretarial efficiency.

Engineers, managers and secretaries all benefit from the advanced editors and tools provided with Intel's systems. Getting the output to a printer is the next step in the process, and is the subject of this application note.

GETTING THE DATA PRINTED

Virtually every computer sold today, from the most inexpensive PC to the largest mainframe, has serial and/or parallel ports for connections to printers and other devices. Intel's development systems are no exception, providing hardware ports for both serial and parallel printer types.

Intel's operating systems supplied with the NDS-II network and development mainframes, INDX and ISIS respectively, provide software "devices" which the user can copy files to. The software device designations are :LP: for the parallel line printer, and :TO: for the serial device. However, varying types of serial printers and their associated protocols render the simple "Copy file to :TO:" inadequate. Additionally, printers are somewhat expensive and noisy. The desired method of operation is to provide one or two printers accessible by a group of people, located in a separate room away from the immediate working area.

This application note shows how Intel's NDS-II network, combined with ISIS Clusters and terminals provide a solution for the desired method of operation. The NDS-II's INDX operating system provides a print spooler that allows users to copy files to a central spool printer (:SP:). Files copied from the remote stations (ISIS Clusters , Series-II/III and Series-IV development systems) are then copied to a parallel line printer connected to the NDS-II.

This print spooling feature is not a new concept for computers, and is only one of many excellent features of the NDS-II system. Many users would like to support additional printers on the network, both parallel and serial, but the NDS-II's built in spooler does not provide for this.

SOLUTION-Prince

Prince is a versatile spooling program designed for use with Intel's Series-II, Series-III, and Series-IV development systems, either in standalone or network mode, and for ISIS Clusters operating with an NDS-II network. Using a dedicated ISIS Cluster is perhaps the most effective and efficient method of operation. The ISIS Cluster solution provides for the cheapest and most automatic operation, which is detailed in Appendix C.

HOW IT WORKS

Prince is an ISIS-based program operating in the 8085 environment of the Development System or ISIS Cluster. After extensive initialization, Prince continually polls the directory that is ASSIGNed to :F8:, and any files in this directory are PRINTED, then DELETED. As this is an ISIS based program, files to be printed must conform to the ISIS file name format:a maximum of six characters, plus an optional three character extension, separated by a period.

:F8: can be assigned to a directory created on a Series-IV for standalone operation, or to a directory on the Network Resource Manager. If the Network Resource Manager is used, and the NRM has no parallel printer attached, you may assign :F8: to /(root)/SPOOL, the main print spooler directory. A workstation could then copy directly to :SP: instead of :F8:. This saves each workstation from having to assign :F8: to a specific directory.

Prince has been designed for optimal use of network resources, and provides additional capabilities and flexibility above and beyond the automatic print spooler provided with the NDS-II. Prince also provides useful capabilities for Series-IV system operating in standalone mode.

Other applications might include operation of a parallel printer at a development system host for ISIS Cluster users, or even communication interface that automatically copies files from one system or network to another system connected via a serial or parallel line.

Upon invocation, Prince automatically checks its environment to determine the type of system it is loaded on. Valid systems are Series-II, Series-III, Series-IV, and the ISIS Cluster. Prince then sets up the appropriate serial channel for output, unless output has been directed elsewhere. For the Series-II and Series-III, this is serial channel 1. The Series-IV uses serial channel 2, and the ISIS Cluster uses the on-board serial channel normally used for the console.

Series-IV systems can use serial printers, but the control interface for the serial device, specifically the XON/XOFF (cntl-s / cntl-q) protocol, is currently not provided with a simple copy to the system serial file (designated :TO:). Prince solves this problem by providing the XON/XOFF protocol, and optionally checks for a hardware printer ready signal if desired, by selectively monitoring Data Set Ready (DSR) on the serial line.

The Intel development systems set the serial channel used for the serial device (:TO:) to a specific file transfer rate, better known as baud rate. Prince can selectively output serial data at user specified baud rates of 110, 300, 600, 1200, 2400, 4800, 9600, and 19200. This allows faster devices and devices that can "buffer up" data to take advantage of the full capabilities of the serial line, while the controls mentioned previously (XON/XOFF and DSR) provide the desired control protocol to run the serial devices and the development systems at their fastest rate.

For management tracking and control, Prince keeps a log of all activity, including error messages, initialization defaults, and information about each file printed. File PRINT.LOG is created in the directory assigned to :F9:, and contains relevant information about the files being printed: the file name, time that the file was printed, owner of the file, and the number of bytes actually printed. The log information can be re-directed to another file, including the console, line printer, or disk file. If the log file specified is a disk file, it can be viewed, copied, or deleted at any time. If the log file is deleted, Prince creates the log file again, using the original log file name, at the next file detected for printing.

Prince allows re-direction of the output to a file rather than the printer connected to the serial line. Spooling to a parallel line printer is accomplished by specifying :LP: as the output path. The output re-direction can also go to a disk file, or any other valid ISIS output file name except :TO:. If a disk file is specified for output, it can be viewed, copied, or deleted at any time. Files being copied to the output file are added to the end of the file. For orderly printing, Prince automatically outputs a form feed before printing each file.

This version is initialized for use with a Diablo 630 serial interface and supports the XON/XOFF protocol at 2400 baud. These parameters may be changed by command line controls.

The ISIS.INI, or submit file that invokes this program, must contain a directory assignment to :F8:, for the files to be printed, and also an assignment to :F9: for the log file, unless it has been re-directed.

The default log file name, if none other is specified, is :F9:PRINT.LOG. Any file specified for the optional re-direction of the log file and/or the output path must be a valid ISIS output file name (refer to the NDS-II ISIS III User's Guide #121765-004 for a definition of valid ISIS output filenames).

INVOCATION AND CONTROL OPTIONS

Invocation of Prince is best accommodated in a command file, or SUBMIT file. For Intel systems, use of a user 'init' file is recommended, and essential for automatic use with an ISIS Cluster. User Init files are automatically submitted for execution upon LOGON to the system. This file contains assignments, and the command line that starts Prince.

Control options are all single letter characters, followed immediately by an "=" sign, then the actual option. Controls can be entered in any order, upper or lower case, can be separated by spaces or commas, but must contain no imbedded spaces. If Output is redirected to a file, as opposed to the default serial channel, then DSR and Baudrate controls have no effect, and the serial channel is not initialized.

Control Description and Examples

```
Controls:                       Control Description:
L=logfile               ; Valid ISIS filename - log file re-direction
                        ; :F9:PRINT.LOG is the default
P=output$file           ; Valid ISIS filename - output re-direction
                        ; can be :LP: for the local line printer, etc.
D=T                     ; DSR control. Any character other than 'T' will
                        ; not set the DSR control - pin 6 on the RS-232
                        ; line is monitored for printer ready. No DSR is
                        ; the default.
B=baudrate              ; valid number. Only the first two characters
   110                  ; are checked to determine uniqueness.
   300                  ; Any following characters are ignored.
   600
   1200
   2400
   4800
   9600
   19200
```

Examples:

1. To set log file to console out and output to line printer:

   :F9:PRINCE l=:co: p=:lp:

2. To set baudrate to 9600 and initialize DSR control (defaults to :F9:print.log):

   :F9:PRINCE b=9600 d=t

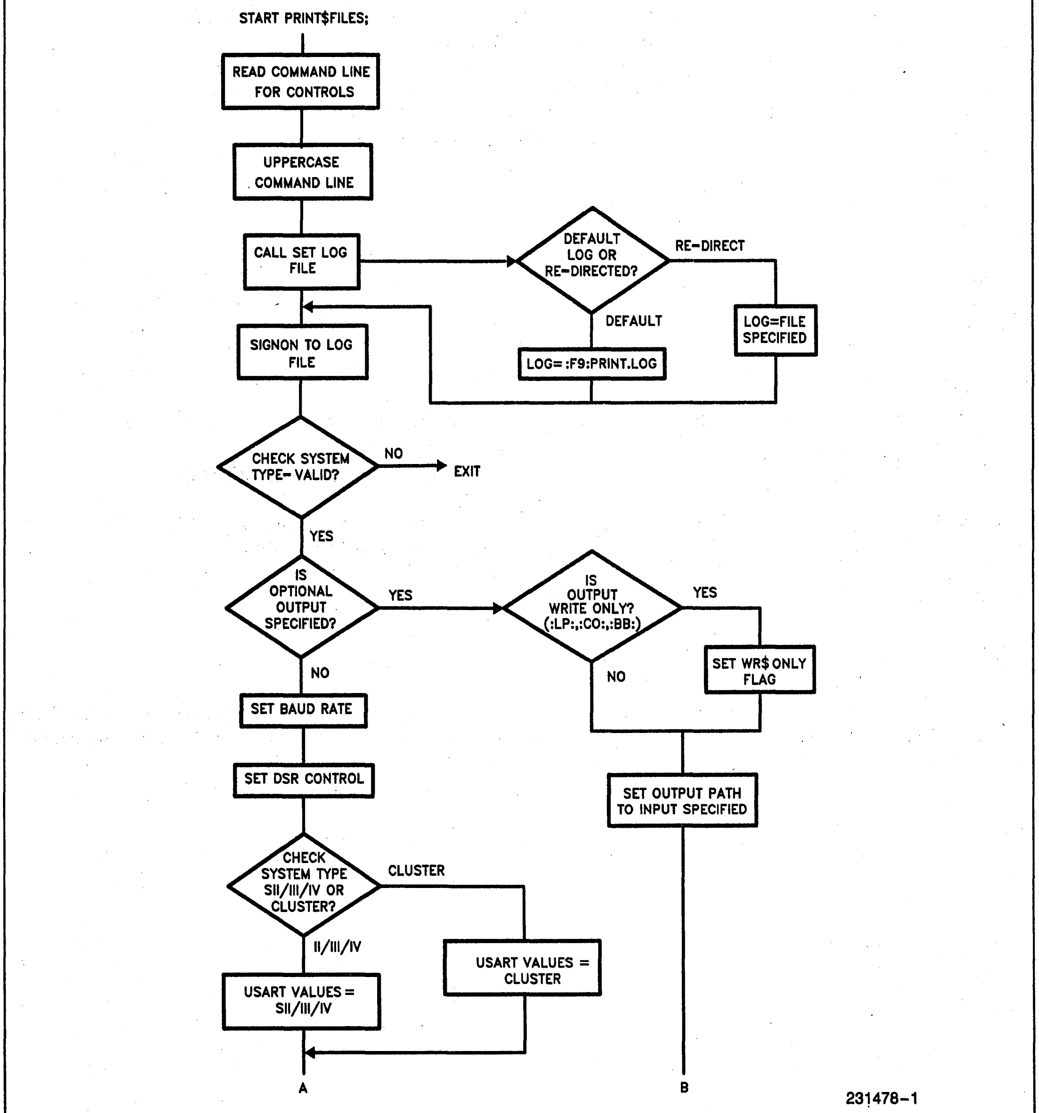
```
Example ISIS.INI:               ; ISIS.INI file for Series-II/III
                                ; and ISIS Cluster
ASSIGN 8 to /w/prntspool.dir    ; copy files to be printed to :F8:
ASSIGN 9 to /w/printlog.dir     ; :f9:also contains the program
ISIS                            ; Invoke ISIS-IV - for Series-IV
:F9:PRINCE                      ; Invoke print spooler
```

CONCLUSION

Prince is a versatile utility that enhances the operation of standalone Series-IV systems or NDS-II networks. Prince is available separately from Intel's INSITE Library, (order PRINTS, Insite order code BG61) and is also available along with many other useful tools in Intel's NDS-II Software Tool Box.

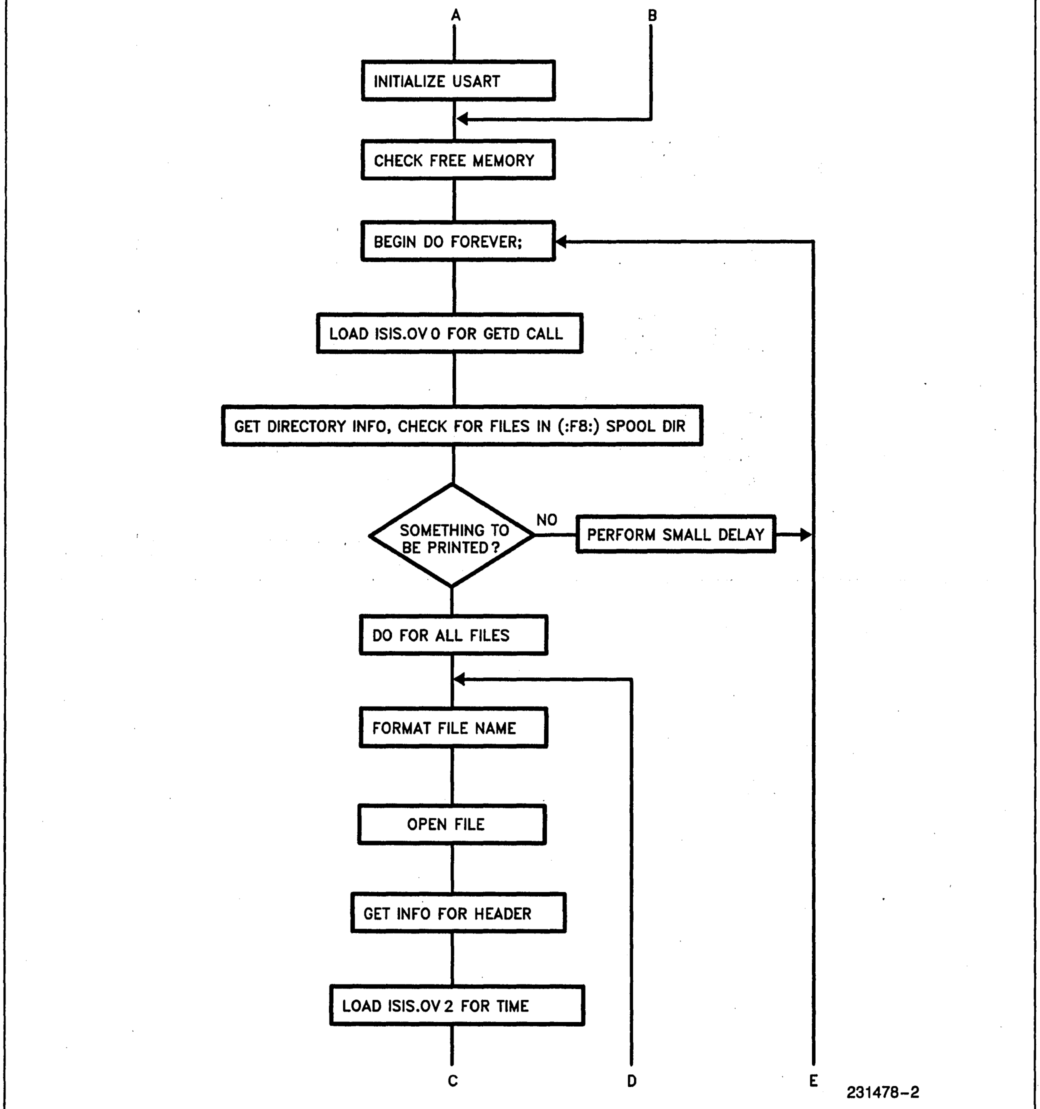
APPENDIX A

PROGRAM FLOW CHART


START PRINT$FILES;
READ COMMAND LINE FOR CONTROLS
UPPERCASE COMMAND LINE
CALL SET LOG FILE
DEFAULT LOG OR RE-DIRECTED?
RE-DIRECT
DEFAULT
LOG=FILE SPECIFIED
LOG=:F9:PRINT.LOG
SIGNON TO LOG FILE
CHECK SYSTEM TYPE- VALID?
NO
EXIT
YES
IS OPTIONAL OUTPUT SPECIFIED?
YES
IS OUTPUT WRITE ONLY? (:LP:,:CO:,:BB:)
YES
SET WR$ ONLY FLAG
NO
NO
SET BAUD RATE
SET DSR CONTROL
SET OUTPUT PATH TO INPUT SPECIFIED
CHECK SYSTEM TYPE SII/III/IV OR CLUSTER?
CLUSTER
II/III/IV
USART VALUES = CLUSTER
USART VALUES = SII/III/IV
A
B

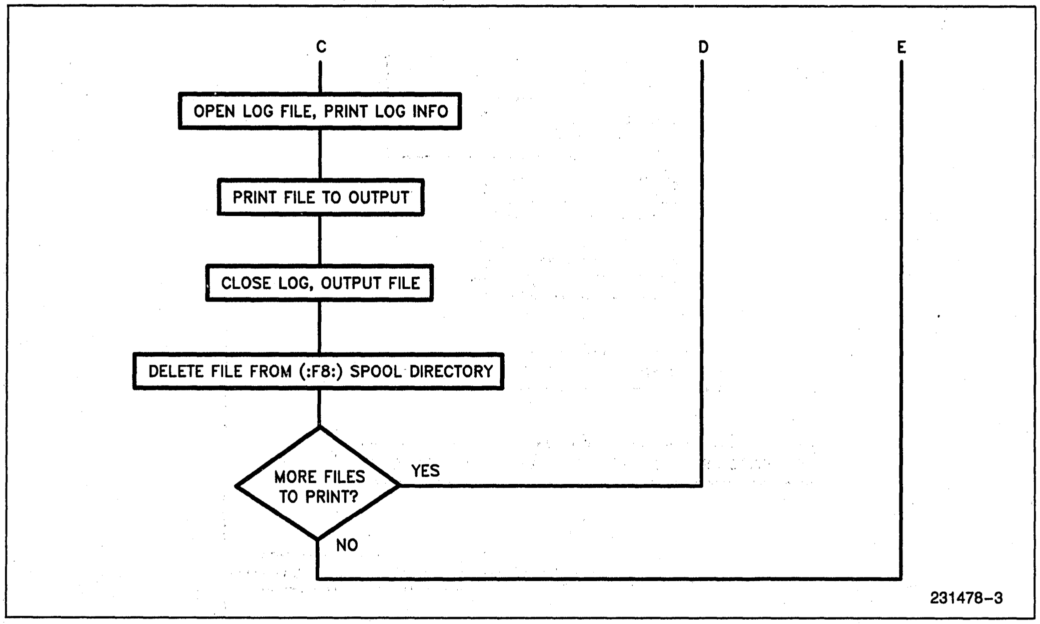
231478-1

PROGRAM FLOW CHART (Continued)



231478–2

PROGRAM FLOW CHART (Continued)



231478-3

APPENDIX B
PROGRAM LISTING

```
PL/M-80 COMPILER    PRINT FILES             PAGE 1
ISIS-II PL/M-80 V4.0 COMPILATION OF MODULE PRINTFILES
COMPILER INVOKED BY:    :fl:plm80 :F3:prince.p80 PAGEWIDTH(80)
               $TITLE ('PRICE') PAGEWIDTH(80)
  1            Prince: do;
     $nolist include(:f3:procs.p80)
     $list
    /****************************** Program Start *************************/
    /*
     Read input line and set log file. Signon to log file, then determine
     system type. Check for optional baud rate control, and DTR/DSR control.
     Then set up the 8251 USART per the system type.
                 */
450  1           call read (1,.input$buffer,128,.in$buffer$actual,.status);
451  1           do i = 0 to in$buffer$actual-1; /* UPPER CASE the input buffer.*/
452  2             input$buffer(i) = set$upper$case(input$buffer(i));
453  2             end;
454  1           call set$log$file;            /* Set up the log file. */
455  1           call print$message(0);        /* Signon to log file. */
456  1           if (high$byte<1) or (high$byte>5) then do;/* Exit if invalid */
458  2              call print$message(11);               /* system type. */
459  2              end;
             /* Check if Line Printer specified in command.*/
460  1           call set$device;
461  1           if lp$flag <> true then do;/* If not line printer, set USART. */
463  2              call set$baud;          /* Set baud rate. */
464  2              call set$dsr;           /* Check for DTR/DSR control. */
465  2              if system$is$SII or system$is$SIV then do;
467  3                 serial$output=.s$serial$output; /* Use SeriesII/IV */
468  3                 wait$for$printer=.s$wait$for$printer; /* serial chn. */
469  3                 end;
470  2              call initialize$usart;
471  2              end;
             /* How much free memory below the Overlay base? */
472  1           limit = 0E87FH - .memory;
473  1           do forever;
             /* Any files to be printed ? */
```

```
474  2          call load(.('ISIS.OVO '), 0, 0, .entry, .status);
475  2          call check$status(1);
476  2          start = 0;
477  2          call getd(8, .start, 1000, .actual, .dir$dump, .status);
478  2          call check$status(2);
479  2          if actual <> 0 then do index = 0 to actual - 1;
           /* Something to be printed */
           /* Format the filename */
481  3               j = 4;
482  3               do i = 0 to 5;
483  4                    if dir$dump(index).filename(i) <> 0 then do;
485  5                         filename(j) = dir$dump(index).filename(i);
486  5                         j = j + 1;
487  5                         end;
488  4                    end;
489  3               if dir$dump(index).filename(6) <> 0 then do;
491  4                    filename(j) = '.';
492  4                    j = j + 1;
493  4                    do i = 6 to 8;
494  5                         if dir$dump(index).filename(i) <> 0 then do;
496  6                              filename(j) =dir$dump(index).filename(i);
497  6                              j = j + 1;
498  6                              end;
499  5                         end;
500  4                    end;
501  3               filename(j) = ' ';
           /* Filename formatted, get the file */
502  3     do while status <> e$file$open;
503  4          call open(.aftn, .filename, read$only, no$line$edit,
.status);
504  4          end;
           /* Get information for the header */
505  3          file$table.aftn = aftn;
506  3          call spath(.file$name, .file$table.device$number, .status);
507  3          call check$status(4);
508  3          call load(.('ISIS.OV1 '), 0, 0, .entry, .status);
509  3          call check$status(3);
510  3          call filinf(.file$table, 1, .file$info, .status);
511  3          call check$status(6);
           /* Load ISIS.OV2 to get the TIME! */
512  3          call load(.('ISIS.OV2 '), 0, 0, .entry, .status);
513  3          call check$status(5);
PL/M-80 COMPILER           PRINT FILES          PAGE 3
           $eject
/* Print the header - form feed to printer, header to log file or :co:*/
514  3     call print(.(FF), 1);
515  3     call open$file$safely (.aftnl,.logfile,wr$only$log);
516  3     call write(aftnl,.header$1, length(header$1),.status);
517  3     call write(aftnl,.filename(4), (j-4),.status);
518  3     call write(aftnl,.header$2, length(header$2),.status);
519  3     call move(4, .zero$time, .dt.system$time);
520  3     call de$time(.dt.system$time, .status);
```

```
          call write(aftnl,.dt.time(0), 8,.status);
          call write(aftnl,.(' on '), 4,.status);
          call write(aftnl,.dt.date(0), 8,.status);
          call write(aftnl,.header$3, length(header$3),.status);
          call write(aftnl,.fileinfo.owner(1), fileinfo.owner(0),.status);
          call write(aftnl, .(cr,lf),2,.status);
          call close (aftnl,.status);
          /* Print the file */
          file$bytes = 1;
          do while file$bytes <> 0;
              call read(aftn, .memory, limit, .file$bytes, .status);
              call check$status(8);
              if memory(0) = FF then memory(0) = 0;
              call print(.memory, file$bytes);
              end;
          /* File has been printed */
              call close(aftn, .status);
              call check$status(9);
              call open$file$safely (.aftnl,.logfile,wr$only$log);
              call write(aftnl,.header$4,length(header$4),.status);
              call print$size (file$info.len$hi,file$info.len$lo);
              call write(aftnl, .(cr,lf),2,.status);
              call close(aftnl, .status);
              call delete(.filename .status);
              call check$status(10);
              end; /* Look for next file */
          /* No files to be printed , Wait a minute or so */
          else do i = 0 to 60;
              do j = 0 to 500;
                  call time(10);
                  end;
              end;
          end; /* of Do forever */
      end Prince;
```

APPENDIX C ISIS CLUSTER BOARD PREPARATION

Used with an ISIS Cluster, Prince can be driven from the ISIS Cluster board's serial channel, which is normally used for a terminal. With the addition of the special SERVER PROM for the ISIS Cluster, the Prince program can be automatically invoked and begin copying files from a network spool directory to a serial printer or other serial device, immediatley upon power-up. In this mode of operation, there is no console connected to the ISIS Cluster. Instead, the serial printer or other serial device is connected to the console port, and the SERVER PROM installed on the Cluster board automatically logs the Cluster board onto the NDS-II network, then submits the ISIS.INI command file. This command file contains the necessary assignments, as well as the Prince program invocation.

To prepare the NDS-II to support the Prince spooler, a username and home directory for the Prince program and SERVER prom must exist. To provide trouble free spooling, the username for the SERVER prom should be declared as a Superuser. This way, file access rights need not be set each time a file must be spooled, printed, and deleted.

The SERVER PROM image is included with the Prince program. The SERVER PROM image can be modified to change the username or password. Bytes 0FF0 to 0FFC (inclusive) are reserved for the SERVER username, password, and string terminator (00H). (Note that these are PROM addresses - this PROM image is moved to a different location in RAM on initialization.)

byte 0FFDH:	PROM checksum
byte 0FFEH:	resreved
byte 0FFFH:	system ID (05 for Cluster - DO NOT CHANGE!)

The following strings are stored in the PROM image:

username: SERVER0<CR>

password: ©

checksum: 082H

If you change the LOGON name and/or password, remember to change the checksum, which is stored in byte 0FFDH. NOTE:The checksum is actually the two's compliment of the checksum calculated by the boot code. Thus, if you change the username to SERVER2 from SERVER1 (increment byte 0FF6H), you must decrement byte 0FFDH. Changing the PROM image can easily be accomplished using Intel's IPPS software, which is supplied with the iUP-200 PROM programmer.

CLUSTER BOARD PREPARATION – PROM BURNING

1. The PROM image is written in 286 format. Remember to initialize the iPPS properly.
2. Read in PROM image, modify LOGON strings, modify checksum, and burn a 2732 or 2732A.
3. Remove the old monitor prom from the Cluster board (A25) and place in the next-door socket (A37) for safe keeping (may be needed by CE).
4. Install new SERVER prom in A25.
5. Install a jumper between pins 67 and 68. This ties Clear-to-Send/ Request-to-Send together on-board. Prince uses XON/XOFF or XON/XOFF and DSR for control, so CTS/RTS is not required between devices.
6. If the printer to be used operates with the hardware DTR/DSR protocol, configure the serial cable such that the printer ready line comes in on pin 6 (DSR) of the serial cable to the Cluster board.
7. Refer to the ISIS Cluster installation manual for further Cluster installation instructions.
8. Set up ISIS.INI file to make assignments and invoke PRINCE, plug it all in and go.

FF0

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
53	45	52	56	45	52	30	0D	0D	00		spare		82	00	05	HEX
S	E	R	V	E	R	0	CR	CR								ASCII

APPENDIX D RELATED PUBLICATIONS

1. ISIS Users Guide 9800306
2. Network Development System II iMDX 450 210937-004
3. CP/M-80 on the NDS-II – Application Note AP 253
4. ISIS Cluster Installation instructions ....

APPENDIX E ERROR MESSAGES

Prince returns messages and error conditions if certain external conditions prevent normal functioning of the spooler. All messages are directed to the log file, unless a fatal ISIS error occurs, preventing Prince from handling the error. ISIS will trap the fatal error and re-boot itself. Some error conditions that are not fatal ISIS errors are considered fatal by Prince, and after logging the error message in the log file, Prince will exit. The 18 messages given by Prince are as follows:

1. Serial printer driver xxx'

   Non-fatal message - The normal sign-on message at Prince invocation.
2. 'ISIS.OV0 not present on system disk'

   Fatal error, Prince will exit. ISIS overlay 0 must be present on :F0:for Prince to function properly.
3. 'GETD system call failed'

   Non-fatal - Prince uses this system call to determine the presence of files to be printed in directory :F8:. Possible causes for failure:a damaged or incorrect ISIS.OV0, file access rights, etc.
4. 'ISIS.OV1 not present on system disk'

   Non-fatal - Prince must load ISIS.OV1 to support the file$info system call. ISIS.OV1 is not on :F0:, or access rights are insufficient.
5. 'SPATH system call failed'

   Non-fatal - SPATH returns information about the file to be printed for log file status of the file.
6. 'ISIS.OV2 not present on system disk'

   Non-fatal - ISIS.OV2 is used to provide time and date information that is placed in the log file for all files printed, and all messages.
7. 'FILINF system call failed'

   Non-fatal - The file$info call returns file information to be used in the log file, such as the owner of the file, etc. If this call fails, meaningful file information will be absent from the log.
8. 'Could not open the file to be printed'

   Non-fatal - Most common causes of this malfunction are insufficient access rights to the file, or an invalid ISIS file name. Rename the file name, or give access rights to the Prince user.
9. 'Could not read the print file'

   Non-fatal - This error will occur only during the printing of the file, before the print has completed, but after the first successful open.
10. 'Could not close the print file'

    Non-fatal - Prince could not close the file just printed.
11. 'Could not delete the print file'

    Non-fatal - Prince attempts to delete the print file after printing. Most common cause of this error is insufficient (delete) access rights. Give Delete Access rights to the Prince user.
12. 'Unknown System Type - not supported'

    Fatal error, Prince will exit. Printfiles checks byte 0FFFFH to discern system type. Valid types are: 01 = Series-II 02 = Series-IV 05 = ISIS Cluster
13. 'BAUD control defaulted to 2400 baud'

    Non-fatal message - An attempt was made to set a different baud rate per the baud rate control (B=number) and number was invalid. Valid numbers are:110, 300, 600, 1200, 2400, 4800, 9600, 19200.
14. 'LOG control defaulted to :F9:print.log'

    Non-fatal message - An attempt was made to redirect the log file, but the log file name was greater than 14 characters. Prince does a gross check on the pathname specified to assure a correct ISIS file name.
15. 'DTR/DSR control activated'

    Non-fatal message - The DSR control is active.
16. 'DTR/DSR control not activated'

    Non-fatal message - An attempt was made to set DSR other than true - DSR not true is the default.
17. 'OUTPUT file defaulted to :F9:print.out'

    Non-fatal message - An attempt was made to re-direct the output file, but the file name was greater than 14 characters. Prince does a gross check on the pathname specified to assure a valid ISIS file name.
18. 'Could not write OUTPUT file'

    Fatal Error, Prince will exit. Prince could not write to the output file specified, so spooling is suspended.

intel®

APPLICATION NOTE

AP-244

October 1985

Distributed Job Control the Key to Increased Network Productivity

SRIVATS SAMPATH
DSO APPLICATIONS ENGINEERING

Order Number: 231480-001

INTRODUCTION

Large software projects and shorter production schedules generate the need for a more flexible and productive development environment, which allows users full access to all available resources.

Recognizing this need, Intel designed the Distributed Job Control (DJC) Facility into the NDS-II system. DJC allows currently idle networked development systems to be supplied to the network as public resources. This is essentially a remote job execution unit to which jobs can be sent by other users on the network. Remote job execution offers higher throughput and increased efficiency, since more than one computer on the network can be controlled and used by a single user. This ability to manipulate idle systems on a network and convert them into productive systems for other users directly translates into increased project productivity.

DJC consists of a set of system utilites that enable the NDS-II system manager to more efficiently run the network. When all idle systems on the network are allocated to other active users, the throughput and efficiency of the network dramatically increases. The Network Resource Manager (NRM) is the nerve center for the distributed job control system (DJC). All jobs are scheduled and queued by the NRM. The NRM also coordinates job cancellation and maintains a system log of job queue activity. DJC, with its powerful set of options, positions itself as an invaluable tool for increased network productivity.

WHY DISTRIBUTED JOB CONTROL?

The need for distributed job control (remote job execution) is apparent in a networked environment where a number of teams are working on different projects. With DJC, all idle systems can be channeled towards the particular time-critical project. As a result, the engineers have control over more than one system and increase their efficiency and productivity.

Figure 1 shows a typical NDS-II system. This network includes the NRM configured with two 84 MB Winchester drives, a 600-LPM line printer, three Series IV Microcomputer Development Systems (one of which has four cluster boards), two Series IIs with one cluster board each, and an assortment of ICETM and I2ICETM modules. Although the development systems are functionally similar, they are logically different as viewed by the NRM. Figure 2 illustrates the difference. Two teams are working on this network. Team 1 is an 8-bit development team, and Team 2 is a 16-bit development team. Both teams are meeting tight deadlines and need all the system time that they can get. One engineer working on the 8086 project is on vacation, as a result, one Series IV is underused. The other two Series IVs do not have anything running in their background. On an average of the 14 computers available to this network, (the Series IVs being counted as two each with foreground/background capabilities), only 10 are being used. The percentage use rate is only 60 percent when it should be close to 100 percent. Percentage use rate can be defined as:

(Total Number of Nonidle Systems/Total Number of Systems)*100

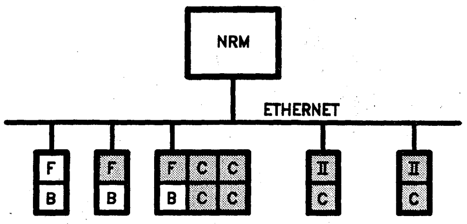
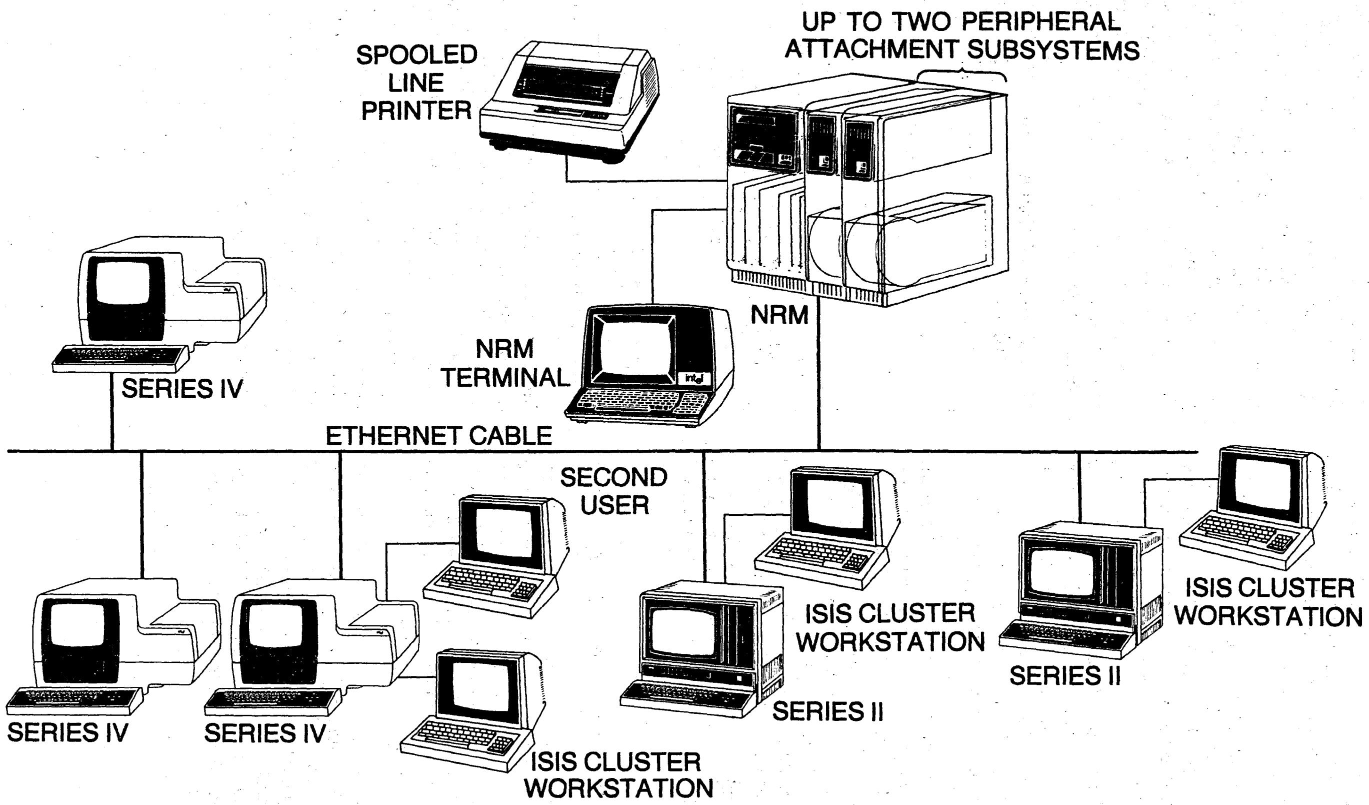


Figure 2. Non-Idle Computers are Shown Shaded

Meanwhile, the 8-bit team is trying to meet a very tight schedule and needs all the system time possible. This team requires a dedicated compile engine that will free its systems for interactive work, such as debugging and editing. DJC can help the 8-bit team by converting the idle machine and the backgrounds of the other two Series IV systems to productive work doers. This enables Team 1 to have all their compiles remotely executed while they concentrate on editing and debugging other modules. These remote execution units can serve both teams, since the Series IV can operate in both 8-bit and 16-bit modes. This results in definite increase in overall team and network productivity.



231480-1

Figure 1. A Typical NDS-II Network

A CLOSER LOOK AT DISTRIBUTED JOB CONTROL

The DJC system recognizes that the network has three types of stations: the NRM, private workstations, and import workstations. The NRM is the nerve center of the DJC system, and it maintains all the state information of remote jobs and status of workstations. A private workstation is one that can send jobs to the NRM and have them executed at other workstations on the network. However, it does not accept jobs from the NRM. These are clasified as work generators. Examples of work generators are Model-800, Series II, Series III, Series IV and ISIS clusters.

An import workstation that can accept jobs from the NRM is called a work doer. Examples of work doers are Model-800, Series II, Series III, Series IV and ISIS clusters. Work generators and work doers use the same hardware. The software executing at the workstation determines if it is a generator or doer. The mix of generators/doers may be flexibly altered through the day/week/project to best suit the user's needs. Normally, when a workstation is first powered up or reset, it configures itself as a private workstation. (The workstation can also be configured to power up as an import station. See Appendix A).

A private workstation can be turned into a work doer for the network withth e IMPORT command. The import command informs the NRM that the private workstation is now capable of doing some type, or types, of jobs. A station remains an import station until the keys CONTROL and C are pressed from its keyboard. If an import station is executing a remote job when the CONTROL and C keys are pressed, it continues executing the job until the job is finished. Only then, does it return to private workstation mode. A Series IV stationthat supports both foreground and background partitions can import into either foreground, background, or both. Thus, a physical station can appear as two import stations to the DJC system.

DJC UTILITIES

Understanding Job Queues

Since the network consists of heterogeneous workstations, some type of mechanism is needed to match the job with the type of workstation it can execute on. This generated the concept of a job queue. A job queue can be envisioned as a waiting place for all work doers and a depository for workgener ators. Job queues are created and deleted using the QUEUE utility, and their statis is monitored using the SYSTAT utility. Each network can have up to 10 queues. The system does not support any predefined queues. While any name may be used, descriptive names based on the work doer's capabilities are recommended. 8-bit.q, 16-bit.q, and print.q are good names, while ONE.queue and compile.queue are not.

The system does not guarantee that a job is sent to a queue capable of doing the job. A Series II or ISIS cluster is only capable of doing 8-bit work. Therefore, the work generator is responsible for ensuring that the work doer chosen can execute the job. Multiple work generators can export to the same queue, and more than one work doer may import from a queue to get jobs done quicker.

Queues are maintained using files at the NRM. The DJC system uses these queue files to maintain job and queue status. These files are shared files and should not be tampered with.

Remote Job Execution

A job is scheduled for remote execution using the EXPORT command. An in depth discussion on the syntax and use of job queues is discussed in Chapter 5 (DJC Utilities). The export command must specify a job queue capable of executing the job. Export checks if the queue exists and displays an exception message if the job is not queued. It also warns the user if there are no importers (work doers) currently serving the chosen queue. The job will wait indefinitely at the job queue until a work doer is assigned to import from this queue.

The exported job may have to wait for some time before it can be executed, since other jobs arrived at the queue earlier might not have been executed. The queue is a first-in-first-out (FIFO) file.

In this way, the DJC system keeps a track of all jobs at the queue and executes them on a FIFO basis.

Example:

```
QUEUE NAME: 16BIT.Q                 QUEUE NAME: 8BIT.Q

JOB NAME        STATUS              JOB NAME        STATUS

FOUR.CSD        WAITING             PRINT.CSD       WAITING
THREE.CSD       WAITING             LOCATE.CSD      EXECUTING
TWO.CSD         EXECUTING           LINK.CSD        DONE
ONE.CSD         DONE                COMPILE.CSD     DONE
```

When it successfully finds an importer for the specified job queue, the NRM will send the job over to that station. At the station, an implicit logon takes place using initialization options that are identical to a normal user logon. For example, the station will take the user's INIT.CSD file and execute it first and then execute the job. The environment set up at the import station is exactly as that at normal logon. The import station, a"reincarnation" of the user who exported the job, has access to all of files the user has. It looks just as if a user is inputting information at the import station. The only difference is that, in this case, input is from a file specified by the user exporting the job.

DJC Commands

DJC system on the NDS-II system has a number of commands that help theuser in effectively configuring an efficient remote job execution system. These are:QUEUE, IMPORT, EXPORT, CANCEL, and SYSTAT.

Each of these commands perform unique and important tasks to make the network distributed job control system a very productive and efficient solution.

QUEUE

QUEUE is a command for managing and displaying job queues at the NRM. The QUEUE command displays the name, number of jobs outstanding and the number of servers. After this information is displayed, the user is prompted to:

ADD DELETE LIST EXIT

ADD	option creates new queues. Up to 10 queues can exist at the NRM
DELETE	option deletes a queue
LIST	redisplays previously displayed information
EXIT	terminates QUEUE

For example:

>QUEUE <cr> will bring up the following display

```
NAME OF QUEUE          # OF SERVERS           # OF WAITING JOBS
16BIT.Q                2                      1
8BIT.Q                 1                      0
PRINT.Q                1                      2
```

Anyone can delete these queues. Since there is no protection offered, the use of the QUEUE command should be restricted to a SUPERUSER. This may be accomplished simply by removing world access rights on QUEUE.86. However, a queue that has jobs waiting cannot be deleted; in this example, only 8BIT.Q can be deleted.

IMPORT

The syntax for the IMPORT command is the following:

```
IMPORT FROM queue ,queue ....... TO
BACKGROUND
```

where

queue	is a character string up to 14 characters long, which names the queues from which the import station can execute jobs. Up to five queues may be specified in the command line.
TO BACKGROUND	is an option that will execute the job in a background mode. This is a Series IV option only.

The IMPORT command declares the given workstation to be a public resource on the network, converting it from a work generator into a work doer. This public resource can now receive jobs from the various queues in the NRM. If the user enters the name of a queue that does not exist at the NRM, an exception message will be displayed. The queues are searched for jobs according to the order in which they were listed in the command line (left to right). If jobs are available on any queues, the import station starts processing them. The importing station starts by performing an implicit logon for the user, whose job is first at the head of the queue. Then it processes the commands within the command file. At the end of the command file, the importing station logs off the user and looks for jobs from the queues to process (left to right).

For example, the import station is configured to execute jobs from 16bit.q, 8bit.q, and iNDXutility.q. Initially, there is only one job on 8bit.q, so execution of it commences at the import station. During theexecution of this job, three more jobs arrive at 8bit.q and one at 16bit.q. The job on 16bit.q will be the next to execute, since the command line in import mode is always scanned left to right.

All output messages from the remote job, displayed on the screen of the import station, may be put in a log file if the LOG option is specified with the EXPORT command. When a station is in import mode, no local processing is possible. To reconvert the import station back to a private station, the user must enter CONTROL-C by pressing both the CONTROL and C keys.

EXPORT

The syntax for the EXPORT command is the following:

```
EXPORT pathname [parameters] TO queue
[{LOG/NOLOG}]
where
```

pathname	is a valid pathname for a command file
parameters	is a list of up to 10 parameters
queue	is the queue to which the job is to be sent
LOG, NOLOG	specifies whether a log is to be kept of all console activity on a mass storage device.

The EXPORT command allows a command file composed at one workstation to be executed on another workstation. The command file must be on a public volume, so that the import workstation can access it. An example of a public volume is a volume resident at the NRM and not a local mass storage device. If the queue does not exist at the NRM, an exception message is displayed and the job does not get queued. LOG, NOLOG determines whether a log file is to be maintained of all console activity, at the import station during the execution of that particular job.

The optional parameters specified in the command line are actual parameters to be substituted for the formal parameters embedded within the command file. In the example below, %0 will be replaced by the name of the source file specified in the command line. This way, one compile command file can handle programs with different names.

In this example, the command file links, and binds a "C" program.

```
Listing for:COMPILE.CSD
cc86      %0.c       debug
link86    %0.obj,                &
          C/sqmain.obj,     &
          C/sclib.lib,      &
          C/small.lib,      &
          C/87null.lib      &
          to %0.86               &
          bind                   &
          ss(stack(+800h),memory(+1200h))
>EXPORT COMPILE(/C--SOURCE--DIR/ISTIME) TO 16BIT.Q LOG
>EXPORT JOB NUMBER :0027H
```

This will export the job to the specified queue (in this case 16BIT.Q), print an export job number, and return control to the user, so that he or she may continue with productive work. Meanwhile, the import station acting as a server for 16BIT.Q will log on as the user, process his or her initialization file, and process the command file COMPILE.CSD. After all commands inCOMPILE.CS D have been processed, the import station goes back into waiting mode and waits for other jobs to be sent from the NRM.

CANCEL

```
CANCEL [BACKGROUND/REMOTE] queue
 {(job name) (# job number)}
```

where

queue	is the queue where the job has been queued for execution
job name	is the final component name of the remote job to be cancelled (in the previous case, the job name will be COMPILE)
job number	is the assigned value of the remote job (this can be displayed by the SYSTAT command discussed next).

The CANCEL command is used to cancel a background or remote job. If the user wants to abort a remote job, the job name and job number must be entered. If the job name is selected and multiple instances of the job name are in the queue, the first one encountered is deleted (this may not be the first one queued). To avoid this, the unique name job number may be used,

Example:

```
> CANCEL REMOTE 16BIT.Q (COMPILE) will
result in
iNDX-W41 (V2.8) CANCEL VERSION V2.8
""COMPILE'' CANCELLED
```

The job name can be substituted with the job number. In this case, it will be 0027H (see example under EXPORT). Once the job is cancelled, the import station will execute the next job in the queue it is serving. If no jobs exist in the queue, it will go into a waiting mode for the next job.

SYSTAT

The syntax for the SYSTAT command is the following:

```
SYSTAT [{QUEUE/MY JOB } (queuename
[,....])] TO PATHNAME [EXPAND] [ALL]
```

where

queuename(s)	designates the name(s) of the queue(s) for which jobs are to be listed
pathname	designates the file where the information is listed
QUEUE	displays information for all queues, or for only those queues explicitly listed after the queue specifier. If this option is specified, the queuenames must be separated by commas.
MYJOB	parallels the queue option but lists information about jobs belonging onlyEXPAND specifies that complete information is displayed for each job. If expand is not specified, condensed information will be displayed for each job.
EXPAND	specifies that complete information is displayed for each job. If expand is not specified, condensed information will be displayed for each job.
ALL	displays appropriate information for all jobs in the specifiedqueue(s). If ALL is not specified, information is displayed only for waiting or executing jobs.

The SYSTAT command is used to display information about the DJC subsystem to the user. There are many options which are best discussed by examples.

Examples:

```
<SYSTAT <cr>
SYSTAT VERSION V2.8
QUEUE          # OF JOBS # OF IMPORT
NAME           WAITING   STATIONS
16BIT.Q        0         1
8BIT.Q         0         1
iNDXUTILITY.Q  1         0
```

This command displays the status of all queues and information on the number of jobs waiting and number of import stations serving any queue. No detailed information of actual job status is shown here.

```
<SYSTAT QUEUE
SYSTAT VERSION V2.8

JOB STATUS FOR: 16BIT.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS
```

No jobs are waiting or executing in this queue.

```
JOB STATUS FOR: 8BIT.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS
```

No jobs are waiting or executing in this queue.

```
JOB STATUS FOR: iNDXUTILITY.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS

PRINTFILE #0028      JOHN       11/30/84   16:20:22     WAITING
```

This command lists by queue all jobs waiting in a queue. This helps in quickly determining the status of jobs in a queue.

```
<SYSTAT QUEUE ALL
 SYSTAT VERSION V2.8

JOB STATUS FOR: 16BIT.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS

COMPILE   #1003      SRIVAT     11/30/84   12:12:30     DONE
COMPILE   #1002      SRIVAT     11/30/84   12:05:19     DONE
COMPILE   #1001      SRIVAT     11/30/84   11:30:20     DONE

JOB STATUS FOR: 8BIT.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS

COMP      #2008      WAYNE      11/28/84   18:12:30     DONE
COMP      #2007      WAYNE      11/28/84   15:10:20     DONE
LINK      #2006      NORI       11/27/84   10:10:23     DONE

JOB STATUS FOR: iNDXUTILITY.Q

JOB NAME  JOB #      OWNER      DATE       TIME         STATUS

PRINT     #2002      JOHN       11/30/84   18:10:20     WAITING
PRINT     #2001      SRIVAT     11/29/84   12:10:22     DONE
PRINT     #2000      WAYNE      11/29/84   10:10:10     DONE
```

This command lists the status of all jobs done or waiting in the queue since the queue was created. This is useful to the system administrator to study queue use.

This command lists the status of all of the jobs that users have submitted. Queue files are circular files 256 jobs long. For example, SYSTAT will display the last 255 jobs done or waiting. If the number of jobs exceeds 256, the first entries (jobs) into the queue file are deleted to make room for the new entries. The expand option, which displays all these jobs, is useful for system administration purposes. Information containing average wait time for each job, the average length of a job, may be obtained. The system administrator may use this information to install another work doer on a particular job queue, thereby optimizing the system for his or her particular environment. This queue can be deleted and then recreated once this information is recorded to clear this log of queue activity.

RECOMMENDATIONS FOR AN EFFICIENT DJC SYSTEM

The following discussion outlines recommendations for a useful DJC system for a network. A number of considerations should be made before your DJC system is implemented on the network.

A minimum of three queues should exist at the NRM:one queue for 8-bit work, one for 16-bit work, and the other an indxutility queue. Normally, one server is enough to serve these queues. However, if the load on any particular application increases, having a dedicated server for that queue will be more efficient.

In the example following, it is assumed that the high 16-bit workload requires a dedicated server for the 16-bit work being done on the network. Therefore, a dedicated server for 16BIT.Q has been generated using the IMPORT command. The other server imports from all three queues. Private workstations can also be converted into import stations whenever they are not being used. The background of one of the private workstations should come up in automatic import mode on powerup. This is discussed in Appendix D.

APPENDIX A
Looping in Export Files

Often, a job needs to be run continuously to do a predetermined task like checking mail. The versatility of the DJC system allows the user to do this in just one submit file. For example, an import station can export a job to itself or any other server on the network.

Example:
```
Mail Box(%0)
Save 1 msg.file
EXIT
checkexist msg.file
if %status ▮ 0 then
     report YOU HAVE MAIL IN BOX %0
end
export mailcheck (%0) to indxutility.q
nolog
end
```

This is an example of an export file that constantly checks for mail in a user's box. If a mail message exists, a message is sent to the user. Checkexist is a program that looks for a specified file and sees the value of %status to 1 if the file exists and 0 otherwise. Report is a utility that sends a message to the user's console. These utilities are explained in depth in the Application Note AP-245:"Using Command Files to speed program development."

The submit file is exported using the command:
```
EXPORT MAILCHECK(SRIVAT) TO
iNDXUTILITY.Q
```

The import station will execute this command file and later reexport the job back to the queue. This job will be put at the end of the job queue behind all others waiting at this queue. It will not totally dominate the job queue. The only way to stop MAILCHECK once it is running is to use the CANCEL command. There is no limit to the number of times an export job can be looped.

Conditional exports can also be done from within an exported job. The IF, THEN, ELSE constructs of command files are used. The above example is just one of the different ways DJC can be used. This feature is very useful if some remote job has to be done continuously.

APPENDIX B
REPORT.86

Since all exported jobs are remotely executed, the only method of monitoring their status is by using the SYSTAT utility. The need for a more interactive status reporter becomes more pronounced. REPORT.86 has been designed to answer this need. REPORT is a utility that should be included in all export files. The syntax for REPORT is the following:

```
REPORT <any message>
```

The following command file example shows how REPORT is used:

```
cc86      %0.c debug                        ; Compile the program
if %status <> 0 than                        ; If error in compile
     REPORT Error in compile of %0.c        ; Send message to user
else                                        ; and exit.
     REPORT Successful compile. Proceeding with LINK
          link86 %0.obj, &
          1/sqmain.obj, &
          1/sclib.lib, &
          1/small.lib, &
          1/87null.lib &
          to %0.86 &
          bind &
          ss(stack(+800h),memory(+2800h))
          if %status ▮ 0 then                            ; Check for error in link
               REPORT Successful Link. End of Job.       ; If no error inform user
          else
               REPORT Error while linking.....           ; If error inform user and
          end                                            ; and exit.
```

REPORT.86 writes the message specified into the user's home directory in a file called REPORT.DAT. All the messages get appended on to this file. The ISIS and iNDX command line interpreters (CLI) have been extended to check for the existence of the file REPORT.DAT in the user's home directory. If the file exists, the contents of the file are displayed on the user's screen. The CLI then deletes this file. This gives the user the ability to constantly monitor the execution of a remote job. In the above example, if there was an error in compilation of the program, REPORT will write the message "Error in Compile of filesheck.c" and the remote job will terminate. This message will then come up on the user's terminal anywhere on the network, and the user can take corrective action. All messages are held until the user returns to the command level. They are not displayed instantaneously in the middle of an AEDIT session, for example.

The REPORT function used throughout the submit file will keep the user constantly informed on the success of all required operations. This results in greater productivity, since the user does not have to wait until the whole submit file is over and then examine the log file. The extensive use of the variable %STATUS in this submit file requires explanation. All Intel utilities, such as PL/M86, C86, and LINK86, exit with a UDI call DQ$EXIT(0) if the operation is successful and DQ$EXIT(n) if the operation was not successful (N is any number). This value passed into the DQ$EXIT call is stored in a variable called STATUS. This variable can be accessed from any submit file. Conditional operations can be done by accessing this variable.

APPENDIX C CHECKTIME.C

Often, a program must be executed at a particular time. CHECKTIME.86 is a utility that allows a program to be executed at a particular time from within a submit file. The concept of STATUS and looping in submit files are used here again. This program obtains from the user a particular time, which can be set to be less or greater than system time. When the defined condition is satisfied, the program will exit with a return code of 1. Otherwise, it will exit with a return code of 0. For example, a match condition will exit with DQ$EXIT(0). This return code is passed on to the %STATUS variable that can be accessed by a submit file.

This program has been designed for doing jobs at a particular time of day in an export file.

The syntax for CHECKTIME.86 is the following:

```
CHECKTIME greater  22:23:45  or
CHECKTIME greater  22:23     or
CHECKTIME greater  22        or
CHECKTIME g        22:23:45  or
CHECKTIME g        22:23     or
CHECKTIME g        22
```

This will return with a return code of 1 if the system time is greater than the time specified and 0 for all other cases.

```
CHECKTIME less     22:23:45
CHECKTIME less     22:23
CHECKTIME less     22
CHECKTIME l        22:23:45
CHECKTIME l        22:23
CHECKTIME l        22
```

This will return with a return code of 1 if the system time is less than the time specified and 0 for all other cases.

For example, backup needs to be done only at a particular time, preferably during the night, when the system load is lighter. This can be done in an export file using the CHECKTIME.86 utility.

```
File: BACKUP.CSD

CHECKTIME g 22:59:00
if %STATUS = 1 then
     TREE BACKUP /APS--WO/USER.DIR/
SRIVAT.DIR/* to /APS1/SRIVAT.DIR
else
EXPORT BACKUP to iNDXUTILITY.Q
end
```

In the above submit file, CHECKTIME compares the given time with the system time. If a match is found, it will exit with STATUS set to 1; otherwise it will exit with STATUS set to 0. If STATUS is set to 0, a match has not been found and the submit file will export itself to the queue. In this way, the jobs get stacked up on the queue. When the CHECKTIME condition does get satisfied, the export file will back up all files in the volume

APS—WO/USER.DIR/SRIVAT.DIR to the /APS1/SRIVAT.DIR.

APPENDIX D Configuring a Station to Come Up as an IMPORT Station

A workstation (Series IV) can be configured to power up as an import station through the SYSGEN command at the NRM. SYSGEN, restricted only to the SUPERUSER, will not allow any other user to modify the system configuration. Invoke SYSGEN by typing:

```
SYSGEN
```

SYSGEN will then clear the screen and display all the workstations on the network and their Ethernet addresses. Select the soft key labelled "Options". Next, select the node that has to come up as an import station. SYSGEN will then display another screen with one of the options being:

(7) Automatic Import to Partition 1 Partition 2

Select the partition needed to to come up in import mode. Both partitions can be selected. SYSGEN will next ask for queue names that will serve that import station. List the queues (maximum of 10) and then exit from SYSGEN. Reset the network and the Series IV will come up as an import station on powerup. To terminate import mode, do a Control-C at the import station keyboard by pressing the Control and C keys simultaneously.

intel®

APPLICATION NOTE

AP-246

October 1985

Setting Up an Efficient Hierarchical File System

WAYNE ROSEN
DSO APPLICATIONS ENGINEERING

Order Number: 231482-001

INTRODUCTION

Software development has become a team activity.

— Team members need an efficient file management scheme.

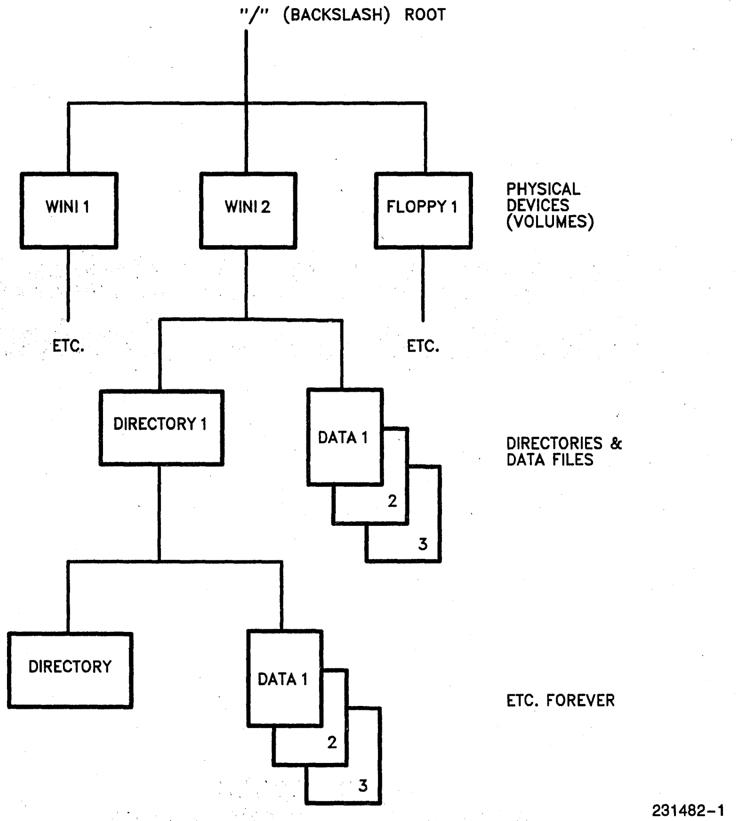
— Team members need to share common databases, but need to be protected against unauthorized file access.

Intel provides a superior hierarchical file system for file management, protection, and sharing in a totally controlled environment.

This Application Note is directed to the NDS-II or Series IV SUPERUSER who is setting up the system's hierarchical file system (HFS) and software development environment. We will be using some hypothetical products to illustrate our recommendations for this HFS.

Intel's NDS-II Network Resource Manager (NRM) and Series IV, running the iNDX operating system, encourage a logically constructed HFS. However, unless set up in a well-structured manner, an HFS can cause many problems. As software tasks grow larger and more complex, a properly structured file system will speed overall system development.

An HFS, (a tree-type file system opposed to a flat file system), promotes system protection and project partitioning and allows users to quickly find needed files. Figure 1 shows a stylized HFS.



NOTE:
1. A "volume" is a mountable, physical device. The system maintains device names for these, which should not be confused with the names you give them. For example:
Device name WFO is the 85 MB Winchester drive controlled by the disk controller board's first driver circuit.
Device name WF1 is the 84 MB Winchester drive controlled by the disk controller board's second driver circuit.

Figure 1. A Stylized HFS

ADVANTAGES OF AN HFS

The following are properties of Intel's HFS (See Figure 1).

- All files have a unique pathname starting from the root.
- Each physical device represents a directory at the root.
- Directories may contain data files or more directories.

Where the root (represented by "/") is the symbolic connection point for all physical volumes of an HFS.

To determine the physical volumes available to you as a user, enter the command "DIR /". For example:

```
DIR /
iNDX-W41 (V2.8) DIR V2.8
DIRECTORY OF /

FILE_NAME LOCATION  ACCESSIBILITY

SYS         remote   ;an NDS-II device
APS0        remote   ;       ""
APS1        remote   ;       ""
WLR.BACKUP  local    ;a local device on
                      my Series IV
```

A Logical Place to Put "Things"

A big advantage to using an HFS is the ability to group files according to user-defined relationships. Let's illustrate this important feature with a story.

We live in a disorganized universe. The laws of entropy tend to maintain and promote this disorganized state. Human beings fight the forces of entropy and try to maintain order in the small niche they carved out for themselves.

In our small corner of the universe, some people like "a place for everything and everything in its place;" that is, they expend some energy organizing their life and surroundings, while others do not bother. Joe Slobotnick (a very bad NDS-II manager) leans towards maximum entropy. Joe does not bother to expend the minimal energy necessary to maintain order on his system.

Joe is the only one in the world who might know where something is kept on the system. Occasionally, even Joe forgets where something needed is stored ("I swear I put that file in the TEMP3 directory along with the other prototypes") At this point, a mad, random-access search begins. This frantic search follows no known rules (like a binary or Shell sort), and no maximum search time can be calculated to tell Joe how long the search will take. Thus, the frantic search may take considerable time, may not be worth the effort, and may have disastrous side effects (like never finding the object of the search).

How much simpler it is for Joe, and for anyone working on his system, if a logical structure is imposed on the system. More importantly, how much simpler for all if a minimal effort is exerted to maintain this logical order.

Protection

Another advantage to using INTEL's HFS properly is file protection. iNDX provides the capability to protect critical files not only from malicious tampering, but from accidental changes (accidents do happen!). For example, all users (even SUPERUSER) should be able to use a compiler; but they should not be able to change or delete it.

Every file in INTEL's HFS has an "owner" associated with it. This owner is someone the SUPERUSER has defined as a system user (see the USERDEF utility). This owner controls access rights to his or her files by:

- Setting the individual access rights
- Setting the world's (the rest of the users on the system) access rights.

Superuser (including those people with secondary Superuser rights) can override the built-in protections and do anything to your files. This is a good reason to restrict the use of Superuser authority to the absolute minimum.

The Software Version Control System (SVCS), Intel's database manager, maintains another level of protection over that provided by the HFS. Features of this utility are discussed in Application Note AP 162 - a PMT tutorial.

Your Home Directory

You will want to keep your personal files in a protected directory that you own. This directory should be your home directory defined at USERDEF time.

When you log on, the iNDX operating system will automatically assign the logical names '' (the NULL logical name) and WORK: to your home directory.

```
LNAME Path
iNDX-W41 (V2.8) LNAME V2.8
LOGICAL      NAME PATHNAME
" '          /volume/USER.DIR/WAYNE.DIR
:WORK:       /volume/USER.DIR/WAYNE.DIR
```

Utilities will default their operations to the NULL logical name if no directory is specified, that is, the DIR command will give a directory listing of my home directory. PLM86 SOMEFILE.PLM will look for the file SOMEFILE.PLM in my home directory. In addition, the NULL logical name is the starting point to easily reference subdirectories located in your home directory. For example:

```
DIR MEMOS.DIR                 ;MEMOS.DIR is
a sub-directory
iNDX-W41 (V2.8) DIR V2.8  ;in my home
directory
DIRECTORY OF /volume/USER.DIR/
 WAYNE.DIR/MEMOS.DIR -full pathname
FILE_NAME    FILE_NAME    FILE_NAME
MANPOW.D14   MAILD.323    UPGRAD.D12
CONF.305     SUNEWS.127   HFS.612
VACATION.N14
```

The NULL logical name can be redefined, but we do not recommend it.

The :WORK: logical name is used by various utilities (including translators) for workspace. We do recommend redefining this logical name. For example, if you are logged onto a network and have an Series IV with a Winchester (a fast device), defining :WORK: to some working directory on your local Winchester will speed you own processing and will reduce overall network Ethernet traffic. In fact, any temporary files created by you should be dispatched to this :WORK: directory. At the end of the day, you can clean up your workspace by simply deleting this working directory.

As you accumulate additional files in your home directory, you should break off related files into subdirectories, such as:

```
MEMOS.DIR          ;all memos
   KEYRESULTS.DIR;those memos that are
key results
```

In fact, we recommend that, other than needed initialization or configuration files, only put other directories in your home directory. Figure 2 is a sample home directory.

Under iNDX, the maximum directory name is 14 alphanumerics. Periods as readability delimeters count as one of the 14 characters.

Under ISIS-III(N), the maximum directory name is 6 dot 3. That is, six alphanumerics, a period and three alphanumerics for the optional extension. Also, it is convenient to name memos in the following form:

```
name.date_code
```

Where date_code = Mdd (three alphanumerics)

M = month code
(1-9 for Jan through Sep, 0 for Oct, N for Nov, D for Dec)
dd = day of the month (01 - 31)

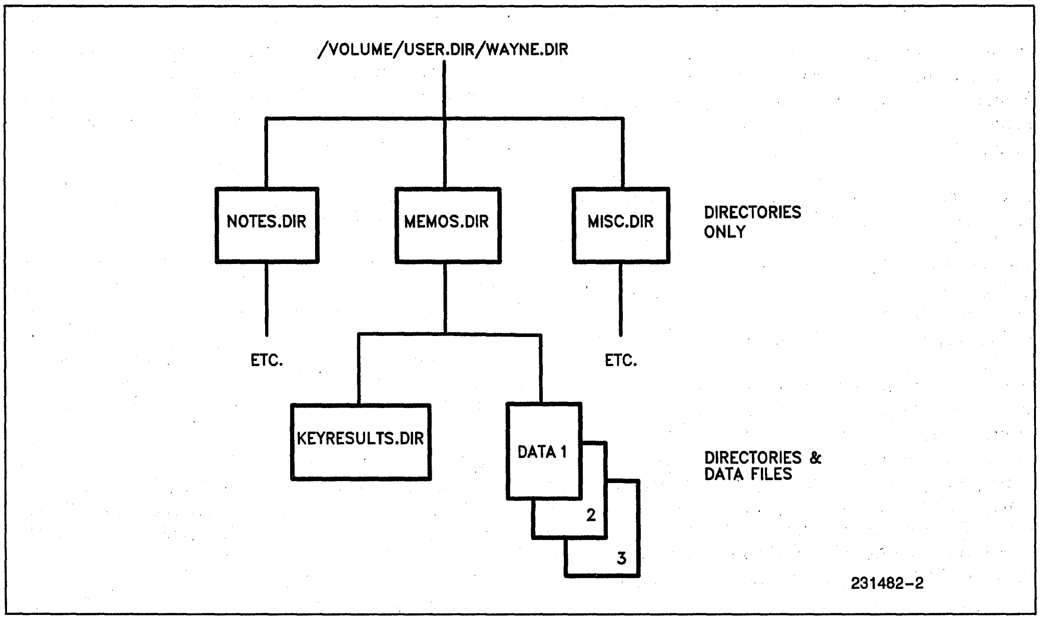


Figure 2. A Sample Home Directory

One Place For Tools

Tools (compilers, linkers, editors, etc.) should be kept in only one location. The world and owner should have DISPLAY ACCESS rights only.

This centralized tool directory is very convenient. Since there is only one copy of each of the tools, the system manager can guarantee that:

- Everyone is using the same version of the tools
- Tool updates need be made in one place only
- When a system generation is done, it can be proven that every module was generated using the same tools.

This last point is very important when it comes to system validation and certification. The Department of Defense (DOD), the Federal Aviation Administration (FAA), and others require such version control guarantees.

What takes place inside the computer when a command is invoked? For example:

```
PLM86 some.file Debug
```

Based on user and system-defined search rules (discussed later in this note), the operating system will begin searching specific directories for the PL/M86 compiler. If the compiler is not found in the first directory, the operating system will then search subsequent directories.

How does the operating system know if a file is in a directory? The operating system performs a linear search through the file entries until a match is found. Files marked as "deleted" and subdirectories count as entries too. The average "match time" is:

($1/2$) $\times$ (total # of file entries) $\times$ (time to perform the match function)

To optimize system performance, you will want to:

- Make certain that the most frequently used utilities are located in the first directory searched
- Order the utilities in this first directory to minimize "match time" for very frequently used utilities,, such as AEDIT, DIR, and COPY (put them into the tools directory first)
- Minimize the total number of files in a directory (especially the tools directory). The maximum number of files that iNDX will allow in a directory is 1,024.

The Problem With Dir

Doing a directory (DIR) listing has to be one of the most frequently used commands of any computer system. However, doing a DIR on an unsorted directory that contains 756 files not only takes a lot of time, but limits the probability of locating all desired files. Your brain and eyes have a difficult time scanning pages of scrolling directory listings. Approximately 75 files (one page of a three-column DIR listing) is the recommended maximum amount of files to be scanned at one time.

How do you pick out subdirectories in a DIR listing? Unless you know the names of those subdirectories, an expanded DIR is the only way to find those directories. Expanded DIRs take a long time and degrade overall network performance. The answer? Where possible, suffix all directories with .DIR. Under iNDX, you have up to 14 characters to specify a directory name. This way, you will be able to find your subdirectories with a standard DIR listing. Or, you can search for occurrences of .DIR only. For example:

```
DIR directory_name FOR *.DIR
```

However, you might prefer having 756 files (or more) in your directories. When you look for a particular file(s), you will use wildcard characters and match for a particular pattern. Unfortunately, this also takes time. Then, there is the problem of possibly missing a needed file that does not quite match the search pattern (or getting other extraneous files that do match the pattern).

The bottom line is this: If you have a system that supports an HFS, use it wisely! And, be sure to:

GROUP RELATED FILES UNDER A
MEANINGFULLY
NAMED DIRECTORY !!

A SAMPLE PROJECT

The following sample project will help illustrate how to set up an NDS-II hierarchical file system. There are many projects being developed on our NDS-II network. The one project our group is working on is:

ROBOTWELDER a dual 186-based project

Our development environment has the following components:

- One NDS-II
- built-in tape cartridge (for back-up)
  - — One 35 MB Winchester (what we originally ordered)
  - — One 84 MB Winchester (we bought this unit when we needed more disk space)
- Two Series IVs
  - — One flippy/winny
  - — One flippy/flippy

- Two Series IIs (our original boxes, pre-network)
- One Series III
- Four ISIS cluster stations
- Intel in-circuit emulators as needed.

A MODEL HFS

Software is divided into three worlds:

*** program equivalent ***

TOOLS: editors, compilers, linkers, etc. (code)

USERS: you, me, and our projects (data)

SYSTEM: network operations (operating system)

In general, the tools operate on output from the users,

Users' files = Tools (users' files)

The system software is responsible for the operation of the computer. The system software manages the tools, the users' files, and itself.

Network operation = System (Tools, users' files, system)

Under iNDX, the system software is responsible for file protection, distributed job control, resource sharing, electronic mail, etc.

Using The Winchesters

Since we have this particular Winchester configuration (see "Sample Project"), we will use the 35 MB Winchester as the boot and system device. In fact, we will make this disk "read only". All of our tools (which have read permission only) will reside on this disk. We get a performance benefit by making this disk read only. A disk write takes approximately seven times longer than a disk read (reduced head thrashing). In our particular configuration, we gain added performance, since there are separate disk controllers for the 35 MB and 84 MB Winchesters (each type of controller can support up to four disks).

Since this 35 MB disk has our tools, contains our system software, and is the boot disk, we will name it something meaningful. For example:

```
/SYS
```

not simply /W or /WO.

Later, we will show why limiting this name to three alphanumerics is useful. An early hint:14 characters is the maximum that the SEARCH CUSP presently accepts.

* Wordstar is a trademark of Micropro.

** Multiplan is a registered trademark of Microsoft Corp.

You should always have at least 2 MB of spare room on the boot disk. iNDX creates many temporary files, some quite large, and puts them on this boot disk. For example, the SPOOL directory is the temporary holding space for print jobs. If you have sent 500k worth or listings (all at once) to be printed, you will need at least 500k free on the boot disk.

Tools

Let us take an in-depth look at these software tools. As far as our NDS-II and workstations are concerned, these tools are divided between:

8-bit tools

- These tools run on the 8085 microprocessor.
- The Series II, cluster board, and Model-800 can run ONLY these tools.
- The hosted 8-bit operating systems are ISIS and CP/M.

16-bit tools

- These tools run on the 8088/8086 microprocessors.
- The Series III and Series IV run these tools (in addition to being able to run all the 8 bit tools).
- The hosted 16-bit operating systems are iNDX and ISIS RUN.

As far as CP/M is concerned, Intel's development tools do not run under CP/M. CP/M is useful if you wish to include others into the development process for example, the professional using Wordstar* and the financial planner using Multiplan.** CP/M running on a workstation on the network is discussed in depth DSO Application Note AP-253:Adding Value to Intel's NDS-11 Development System Network with Network CP/M-80.

It is a misconception to believe that tools running on a 8-bit machine can only generate objects that an 8-bit microprocessor can use. Intel supplies a 8-bit PL/M compiler (i.e., runs under ISIS) that generates object code for an 8088/8086 microprocessor.

Due to the inertia of history or tradition, the 8-bit world is called:

```
ISIS.SYS (it would have been nice to
call it 8BIT.DIR).
```

However, we can call our 16-bit world:

```
16BIT.DIR.
```

16BIT.DIR

It is useful and very convenient to subdivide ISIS.SYS and 16BIT.DIR into logical groups. Under ISIS, we have great flexibility to do this. A Series IV can specify one additional Search path right now.

We would like to digress a bit and talk about the iNDX SEARCH CUSP. Under ISIS, when a CUSP is invoked or referenced by just its name, the command line interpreter (CLI) looks for the CUSP in the default directory, :FO:. For example:

```
 DIR (this is the same as typing
:FO:DIR)
```

Similarly, under iNDX, when a CUSP is invoked or referenced by just its name, the CLI:

- First looks in the system volume directory (in our example, this is called /SYS).
- If not found, the CLI then looks in the directory specified by the NULL logical name (''). The NULL Logical name is defaulted to your home directory and should not be changed.

It is convenient to have additional search paths. The current SEARCH CUSP gives us one more, which we can use to point to 16BIT.DIR:

```
SEARCH /SYS/16BIT.DIR
```

Thus, for our Series IVs, our search paths are:

```
 /SYS/16BIT.DIR
 /SYS                           ;boot
device
 /WORK/USER.DIR/home_directory ;the
NULL logical name
```

The CLI will first search /SYS/16BIT.DIR, then the boot device, and finally our home directory. A CUSP found in any of our search directories with an entry in the menu compiler, will be able to:

— Access its syntax builder

— Complete command lines (FILL ON)

— Display HELP messages for any portion of the command line.

Why did we limit the system volume name to three alphanumerics? Currently, the search pathname, /SYS/16BIT.DIR, cannot be longer than 14 alphanumerics (including backslash delimeters). Our way to get around this limitation (if you have a longer volume name) is to use an LNAME.

```
  LNAME define 16BIT.DIR for
/long_volume_name/16BIT.DIR,
```

But then:

— You use one more LNAME

— This LNAME cannot be removed or redefined

— All users have to set up this LNAME.

ISIS.SYS

It seems that everyone sets up his or her own virtual floppy assignments in individual ISIS.INI files. We recommend that the group adopt a common standard and stick with it. We suggest the following:

```
                        *** 8-bit workstation ***
ASSIGN    :FO:to /SYS/ISIS.SYS
ASSIGN    :F1:to :F9:today's_project.DIR (your working directory)
ASSIGN    :F2:to /work_volume/PROJECT.DIR/xxx.DIR/DATABASE.DIR/MODULE.dir
      ;xxx is the project you're working on
      ;module is the particular piece of the project you're working on at the
       moment (if a large project)
ASSIGN    :F3:to /SYS/ISIS.SYS/LIB.DIR
      ;libraries, system $INCLUDE files
ASSIGN    :F4:to /SYS/ICE.DIR/which_ICE_you're_using.DIR
      •
      •
      •
ASSIGN :F9:to /work_volume/USER.DIR/home_directory.DIR
```

NEVER, NEVER re-ASSIGN :F9:, your home directory.

16-BIT WORKSTATION (SERIES III)

This is a Series III in RUN mode. In addition to the assignments above, add the following:

```
        ASSIGN    :F7:to /SYS/16BIT.DIR/LIB.DIR       ;16-bit libraries
        ASSIGN    :F8:to /SYS/16BIT.DIR               ;16-bit CUSPS
```

DIRECTORY STRUCTURE FOR TOOLS

We will put all interactive software tools under one of the following directories:

```
/SYS/ISIS.SYS, or
/SYS/16BIT.DIR
```

based on whether the tools are hosted on an 8-bit or a 16-bit processor.

Since commonly used $INCLUDE files (.H files for you "C" people) and libraries are nonexecutable and are usually brought in during a SUBMIT file, we can put them into subdirectories:

```
/SYS/ISIS.SYS/LIB.DIR, and
/SYS/16BIT.DIR/LIB.DIR
```

Project-related $INCLUDE files should be stored in the project database.

There are other files that have nothing to do with the host processor, such as configuration files that set up a terminal for an editor or PSCOPE or configure a terminal for a second user (Series IV). These we will put under:

```
/SYS/CONFIG.DIR
```

Our target processor (the processor(s) in our product) has nothing to do with the host processor that our development system is running. For this reason and for modularity and partitioning purposes, we have elected to break out all of the ICE emulator software and lump it under a separate directory:

```
/SYS/ICE.DIR
```

For convenience, certain CUSPs should be kept in the root directory of the boot device. We suggest that all you need to leave behind are:

```
LOGON    ;Never remove from the root
(see Appendix A)
DIR.86
LOGOFF.86
```

You may be wondering why we chose to remove as many CUSPs out of the root as possible. The root directory of the boot device is already cluttered with many system files; the total number can be substantial. (See Appendix 1 for a list of these system files.)

Figure 3 contains our suggested directory structure for tools.

```
/SYS                                                 ;volume name
          /16BIT.DIR                                 ;16-bit tools
                    /LIB.DIR                         ;libraries
          /ISIS.SYS                                  ;8-bit tools
                    /LIB.DIR                         ; libraries
          /CONFIG.DIR                                ;configuration files
                                                     ; for various terminals
                    /AEDIT.MAC.DIR                   ; AEDIT
                    /STTY/CFG.DIR                    ; Series IV users
          /ICE.DIR                                   ;in-circuit emulators
                    /ICE51.DIR
                    /ICE.86.DIR
                    /I2ICE.DIR
                         •
                         •
                         •
```

Figure 3. Directory Structure for Tools.

The first thing your INIT.CSD (Series IV LOGON initialization file) should do is:

```
SEARCH /SYS/16BIT.DIR
```

THE CASE OF THE MISSING CUSPS

You have looked everywhere on your disk, but you simply cannot find the EXPORT.86 file. Has someone deleted it? No! This iNDX CUSP, and other "hidden" CUSPs listed below, are built directly into the iNDX CLI.

***COMMAND FILE PROCESSING

BATCH	command file editor with syntax help
SUBMIT	executes the command file instantaneously
EXPORT	executes the command file at another time and place
BACKGROUND	executes the command file in my background now

***COMMAND FILE CONTROLS

COUNT	allows multiple executions of commands
REPEAT	allows multiple executions of commands
UNTIL	used by COUNT and REPEAT
WHILE	used by COUNT and REPEAT
ENDJOB	terminates this command file now
OPEN	opens a parameter file
READ	gets a parameter from a file
SET	allows (re-)definition of CLI variables
IF	conditional execution
ORIF	conditional execution
ELSE	conditional execution

***MISCELLANEOUS CUSPS

SEARCH	enables or lists CLI search paths
FILL	enables disables CLI command completion
LOG	saves all console output to a file
END	noop command for ISIS compatability
RUN	noop command for ISIS compatability
VIEW	scan a file (AEDIT-like interface)

The command file controls (IF, UNTIL, etc) are very useful for controlling command file (SUBMIT) execution. (Refer to the Application Note AP 245 for further discussion.)

Users

The next part of our software world is for the users. The first part of this world contains all our personal, home directories. The second part of this world contains all the project files.

The following example shows setting up this directory. We are using our 84 MB Winchester as our work disk and, therefore, are naming it WORK1.

PEOPLE

```
/WORK1                                              ;volume name
     /USER.DIR                                      ;main directory
               /SUPERUSER.DIR                       ;with release 2.8 of iNDX, the
                                                     superuser
                                                    ;gets his or her own home directory
               /WAYNE.DIR
                    /MEMO.DIR                            ;as an example
                         /KEYRESULTS.DIR                  as an example
                         /APNOTES.DIR                    ;as an example
                    /BRIAN.DIR
                    /CHRIS.DIR
                    /DEBBIE.DIR
                         •
                         •
                         •
```

Figure 4. Setting Up Home Directories

USER.DIR contains all the home directories for everyone using the system. These directories should be assigned at USERDEF time;

```
    USERDEF define WAYNE id 20000 DIR
/WORK1/USER.DIR/WAYNE.DIR
```

The names of the home directories should be the usernames (the name asked for at logon time) plus the suffix .DIR.

NOTE:
For normal operation, I will logon as WAYNE. When superuser priviledges are required, I can logon as SUPERWAYNE (previously defined as a secondary superuser). If you reserve logging on as SUPERUSER for the times you need to do a USERDEF, you can let the system protect you as it was designed to do.

PROJECTS

The next major directory is for our projects. Our group is working on ROBOTWELDER. Other groups are also using the NDS-II. Lump their project directories under PROJECT.DIR, too.

```
/WORK1                                      ;volume name
     /PROJECT.DIR
          /ROBOTWELDER.DIR                  ;a main project
               /DATABASE.DIR
                    /SYSTEM.DIR             ;overall system database
                     Only put SVCS-type files in this directory.
                    /ASM.DIR
                     Only put SVCS-type files in this directory.
                    /DISPLAY.DIR
                     Only put SVCS-type files in this directory.
                    /database4.DIR
                     Only put SVCS type files in this directory.
                         •
                         •
                         •
          /yet--another--project.DIR
               /DATABASE.DIR
                    /SYSTEM.DIR             ;overall system database
                     Only put SVCS-type files in this directory.
                    /database.2.DIR
                     Only put SVCS-type files in this directory.
                         •
                         •
                         •
```

Figure 5. Project Directories

Put all files associated with your projects into a protected SVCS database. These file include source and objects, $INCLUDE files, MAKE files, and documents.

Break the database into many databases, each supporting a particular function or system block. In our example for the ROBOTWELDER project, one major system building block is called DISPLAY. We lump all files connected to DISPLAY into a subdirectory. DISPLAY is just one basic function of our slick new microprocessor-based ROBOTWELDER machine.

Each database should contain no more than 50-related modules to reduce database contention. The system database contains the files associated with overall project maintenance and organization. These files include block diagrams, documents and memos, timetables, and system integration test procedures.

You should not keep .LST files in a database or even on the Winchesters. They take up alot of space, and can always be regenerated when needed.

NOTE:
Only people using the network should have user names. Do not set up a user name, for example, called ROBOTWELDER.
We believe that if you do set up a project user name, such as ROBOTWELDER, with people logging on with this user name, you will lose control over your sources. It will be difficult to know who made changes on files. (Refer to the Applications Note AP 162 for further discussion.)

APPENDIX A

SYSTEM FILES

Appendix A contains a list of description of those files that the iNDX operating system maintains in the boot disk. Files beginning with r?DUP are duplicate files maintained by the operating system in case a disk "glitches." The original files are kept near the physical front of the disk, and the duplicates are kept near the back of the disk.

If the operating system detects that an original file is bad, a warning message will be printed and the duplicate files will be used. At this time, save all your files onto another device and reFORMAT the suspect disk. Otherwise, you might lose everything on your disk.

CAUTION: unless you really know what you are doing:

LEAVE ALL THE FILES LISTED BELOW ALONE!!!

DJC Queues

1. Information about DJC queues. These file can grow to contain a maximum of 256 job entries.

```
        16BIT.Q        ;an import queue we created for 16-bit jobs
         8BIT.Q        ;an import queue we created for 8-bit jobs
           etc
```

HINT: Give your queues meaningful names, not like:

FOOWAFFLE or BOZO.

2. DJC header file.

```
        DJC--CHK--PT   ;contains names of queues, which stations service
                       ;which queues, and the protocol version number
```

Series IV Temporary Files

1. SUBMIT jobs.

```
    88--CMD--18          ;as an example
    88--CMD--19          ;the naming format is 83--CMD--xy
        •
        •
        88--CMD--Y9
        etc.

        and

    88--STACK--18   ;as an example
    88--STACK--19
        •
        •
    88--STACK--Y9
        etc
```

2. LNAMES (logical names the Series IV people are using)

```
    88--LNAME--1
        •
        •
    88--LNAME--J
        etc
```

Series IV and NRM CLI File

```
      CLI_HELP                         ;large Series IV text file
      PRM_HELP                         ;large NRM help text file

      CLI_SYN_TBL                      ;used by the Series IV menu compiler
      PRM_SYN_TBL                      ;used by the NRM menu compiler

      HCLI                             ;Series IV error messages
      PRM_HCLI                         ;NRM CLI error message
```

Series IV and NRM Logon Files

```
  ***Series IV
      LOGON                        ;logon CUSP
                                   ;delete this and no one can logon to Series IV
      LOGON_HELP                   ;online HELP text
      LOGON.SYN                    ;logon menu line

  ***NRM
      PRM_LOGON                    ;logon CUSP
                                   ;delete this and no one can log on at NRM)
      PLOGON_HELP                  ;online HELP text
      PLOGON.SYN                   ;logon menu line
```

Electronic Mail

1. MAIL.DIR is a directory. In this directory, electronic mail will set up individual mailbox directories.

```
      MAIL.DIR
```

As an example:

```
          WAYNE                   ;all of Wayne's messages will be put in this
                                  ;directory
          BRIAN                   ;Brian's mailbox (MAIL uses USERDEF usernames)
```

iNDX Operating System

```
    OS88.RESIDENT               ;NRM operating system, iNDX.G11 (no overlays)
    OS88.OVERLAY                ;empty (length 0)
```

Communication Software

```
    SYSTEM                      ;a directory that contains the following files
        CONFIG                  ;SYSGEN info (including Ethernet addresses)
        MUSER.INFO               terminal configuration info
        COMMIDOS                ;communication info
        COMM3.X02               ;Ethernet communication software
        COMM3.X03               ;Ethernet communication software
        INDX.W31                ;OS downloaded to a SeriesIV/3
        INDX.W41                ;OS downloaded to a Series IV/4
```

Disk Maintenance

```
    VERIFYFIX                       ;a directory used by the VERIFY CUSP
    r?BADBLOCKMAP                   ;which parts of the disk not to use
    r?DUPBADBLOCK                   ; duplicate
    r?SPACEMAP                      ;which parts of the disk are used
    r?DUPSPACEMAP                   ; duplicate
    r?FNODEMAP                      ;which directory entries are used
    r?DUPFNODEMAP                   ; duplicate
    r?DUPFNODE                      ;contains all file information (redundant)
                                    ; Note:an iNDX disk is RMX-86-compatible
    r?DUPBLOCKZERO                  ;a copy of the first block on the boot disk
    r?VOLUMELABEL                   ;name of the disk
```

System Files

```
UDF                         ;user-definition file (NAMES + PASSWORDS)
BAK.UDF                     ;a duplicate copy you should make every time
                            ; you do a USERDEF
HOME                        ;user names and home directories
BAK.HOME                    ;a duplicate copy you should make every time
                            ; you do a USERDEF
PUBLIC.UDF                  ;public versions of UDF NAMES and home directories
BAK.PUBLIC.UDF              ;a duplicate copy you should make every time
                            ; you do a USERDEF
SPOOL                       ;the directory behind :SP:device
r?ACCOUNTING                ;not currently used
r?ISOLABEL                  ;standard ISO label
r?RESERVED1                 ;reserved for future use
r?RESERVED2                 ;reserved for future use
```

APPENDIX B

Acronyms and Definitions

iNDX	(i)ntel (N)etwork (D)istributed e(X)executive Intel's proprietary 16-bit operating system that runs on the NRM and the Series IV.
ISIS	(i)ntel's (S)ystems (I)mplementation (S)upervisor Intel's proprietary 8-bit operating system that runs on the Model-800, Series II, Series III, Series IV and cluster stations.
RMX	(R)eal time (M)ultitasking (E)xecutive Intel's real-time proprietary system (8-bit and 16-bit versions).
CLI	(C)ommand (L)ine (I)nterpreter
CUSP	(C)onnonly (U)sed (S)ystem (P)rogram
NDS-II	(N)etwork (D)evelopment (S)ystem, version II
NRM	(N)etwork (R)esource (M)anager
PL/M	(P)rogramming (L)anguage for (M)icroprocessors Intel's system's implementation language.
PMTs	(P)rogram (M)anagement (T)ools
SVCS	(S)oftware (V)ersion (C)ontrol (S)ystem, one of the PMTs An automated means of tracking changes to program source code, maintaining variants of the source and objects modules for a program, and recording access to the source and object modules in a multiprogrammer environment.
MAKE	not an acronym, one of the PMTs A program designed to generate a submit file that can be used to construct the most current version of the requested software.

intel®

APPLICATION
NOTE

AP-247

November 1986

Adding Capability to the NDS-II System with Cluster Boards

CHRIS FEETHAM
DSO APPLICATIONS ENGINEERING

Order Number: 231483-001

INTRODUCTION

The ISIS cluster board (iMDX 581) was introduced into the NDS-II product line to reduce dramatically the cost of a personal workstation. It achieved this goal and gave the network numerous expansion opportunities. All of the applications discussed in this note are available through the NDS-II toolbox.

ADDING ADDITIONAL USERS

The cluster board is a single MultibusR board with an 8085-2 processor, 64 K of RAM, an RS232 serial port, and other supporting circuitry. Figure 1 shows a block diagram, and a complete circuit diagram is included in Appendix A. A cluster board may be installed into any master slot of a network Model 800 or Series II, III, or IV development system to support an additional network user via a dumb terminal. This low-cost method of adding extra users to the network served as the primary motivator for the development of the cluster board.

With the exception of Multibus slot, some power, and access to the host's Ethernet controller board, the cluster board uses none of its host development system's resources. The cluster board does not slow the host, which generally has no knowledge of its presence in the system. A host may support multiple cluster boards. Figure 2 shows the maximum number that may be added to each host system.

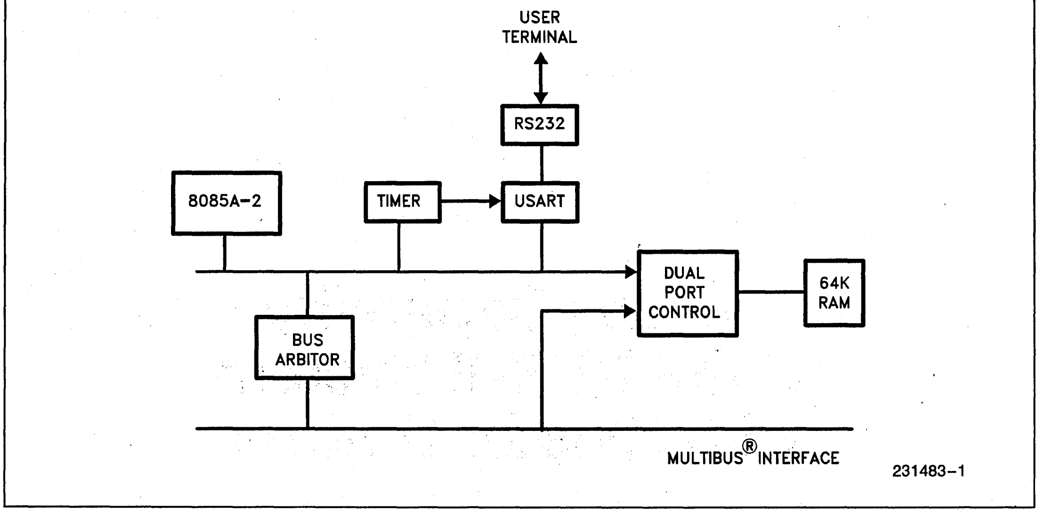


Figure 1. Cluster Board Block Diagram

During initialization of the host system, an operating system is loaded from the network resource manager (NRM) into the RAM of the cluster board. While ISIS operating system was chosen to ensure compatability with previous development environments, CP/M-80 may also be used (see AP 253). During operation, ISIS accesses data files and programs from the protected hierarchical file system of the NRM using the Ethernet controller boards. Access to local host devices, such as floppy disks or Winchester disks, is not permitted.

In normal use, a dumb CRT would be connected to the RS232 port of the cluster board. The user would then have access to all of the 8-bit network tools, including full-screen editors, program management tools, and electronic mail. While some 8-bit compilers are also available, the cluster board is generally used for interactive work supportng the engineer (or the support staff). Access to 16-bit advanced tools is available via the Export facility of the networks' distributed job-control system, where the cluster user may generate a job using local tools and then request its execution on a more capable system upon the same network. This productive shared-tool environment is described further in AP 244.

It is not mandatory to install a dumb CRT. In fact, any RS232 device will suffice. The possibilities are endless, since RS232 is one of the few standards in the electronics industry today. Although this article will discuss various applications, the solution is general in nature, and anysystem with an RS232 interface could be connected to the cluster board.

REMOTE NETWORK OPERATION

Figure 3 shows the connection of an Intel iPDSTM portable development system. The iPDS system is especially suited to 8-bit microprocessor applications development. It has many tools for individual development but does not include advanced network tools, such as electronic mail or program management. In this application, the iPDS system is at a remote site, and a modem link connects the iPDS system to the NDS-II network.

A dumb terminal emulator program called REMOTE has been written for the iPDS system. This program, as part of the network toolbox, includes autodialing a Hayes smart modem. While running in terminal emulation mode, the iPDS can access all facilities of the network, including electronic mail and distributed job control facilities. REMOTE also includes a file-transfer protocol that enables data transfer between the iPDS system and the NRM.

If the iPDS system is at a service location, you need a diagnostic program from the NRM. Or, the iPDS could have data gathered from a remote site to be analysed back at base. The possibilities are endless.

System	Maximum Clusters
Model 800	2
Series II	3
Series III	1
Series IV	3
Expansion Chassis	4

Figure 2. Adding Cluster Boards to Host Systems

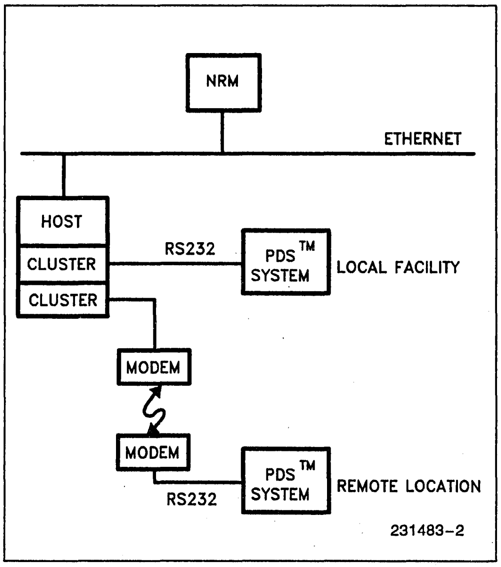


Figure 3. Attaching the iPDS™ System to the Network via an ISIS Cluster Board

ADDING AN ADDITIONAL PRINTER

An additional printer is often required on an NDS-II system. Letter quality printers are popular and their RS232 connection makes them a natural for connection to the cluster board. One problem - how does an output device such as a printer LOGON to the network and initiate file transfer from file to paper.

Server is a slight modification of the standard cluster PROM - it includes a PROM based console to solve the initialization problem. After power-up the LOGON program calls the console input routine to input the user name and password - within server a user name and password is supplied from PROM (Refer to the AP-242 — Additional printer support for the NDS II — for more complete information.)

Once logged on the system executes an initialization file ISIS.INI from the users home directory. In this server example a program that never exits will be chosen - PRINCE, a versatile serial printer driver, is such a program. Following initial drive assignments PRINCE polls a directory looking for files, once a file is identified it is copied to the serial printer and then deleted - simple but most effective.

AUTOBOOT CLUSTER BOARD

BOOTUP is an extensively modified cluster PROM. Rather than rely upon a host system to provide its operating system BOOTUP allows a cluster board to load its own ISIS operating system from the network. Following power-up BOOTUP initializes an SBC550 Ethernet controller and then logs on to the NRM under a predefined name of ISIS. Once logged on BOOTUP loads its operating system from the network. Before passing control to the user BOOTUP seeks out and initializes any other cluster boards also installed within the same chassis.

BOOTUP provides the network user with a low cost method of adding software developers - take any iSBC chassis, add an Ethernet Comm set and a cluster board containing the BOOTUP PROM and the system is complete. Up to seven additional cluster boards may be added to provide a very low cost eight-user environment as shown in Figure 4. BOOTUP also supports the server concept. The BOOTUP PROM is provided with the Network/Series IV Toolbox product.

CONCLUSION

I hope I have explained some of the versatility of the ISIS cluster board. Think of it as a universal interface board between the complex multi-protocol world of Ethernet and the straight forward start-data-stop world of RS232. I am sure this will prompt many new applications for the product - feel free to experiment and benefit from your findings.

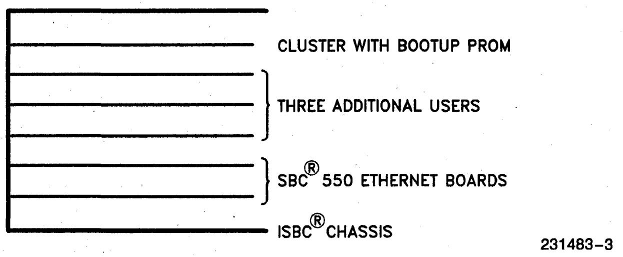
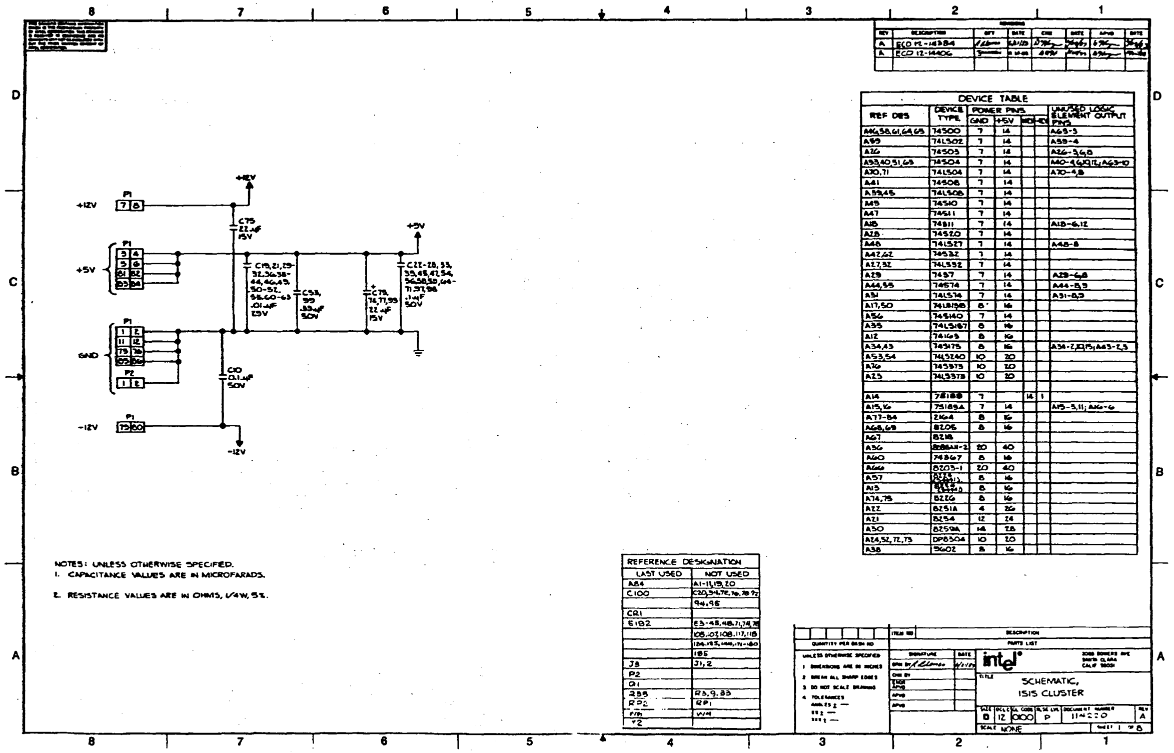


Figure 4. BOOTUP Allows a Low Cost iSBC® Chassis to Act as a Host for Software Developers

APPENDIX A
CLUSTER BOARD ADDS CAPABILITY

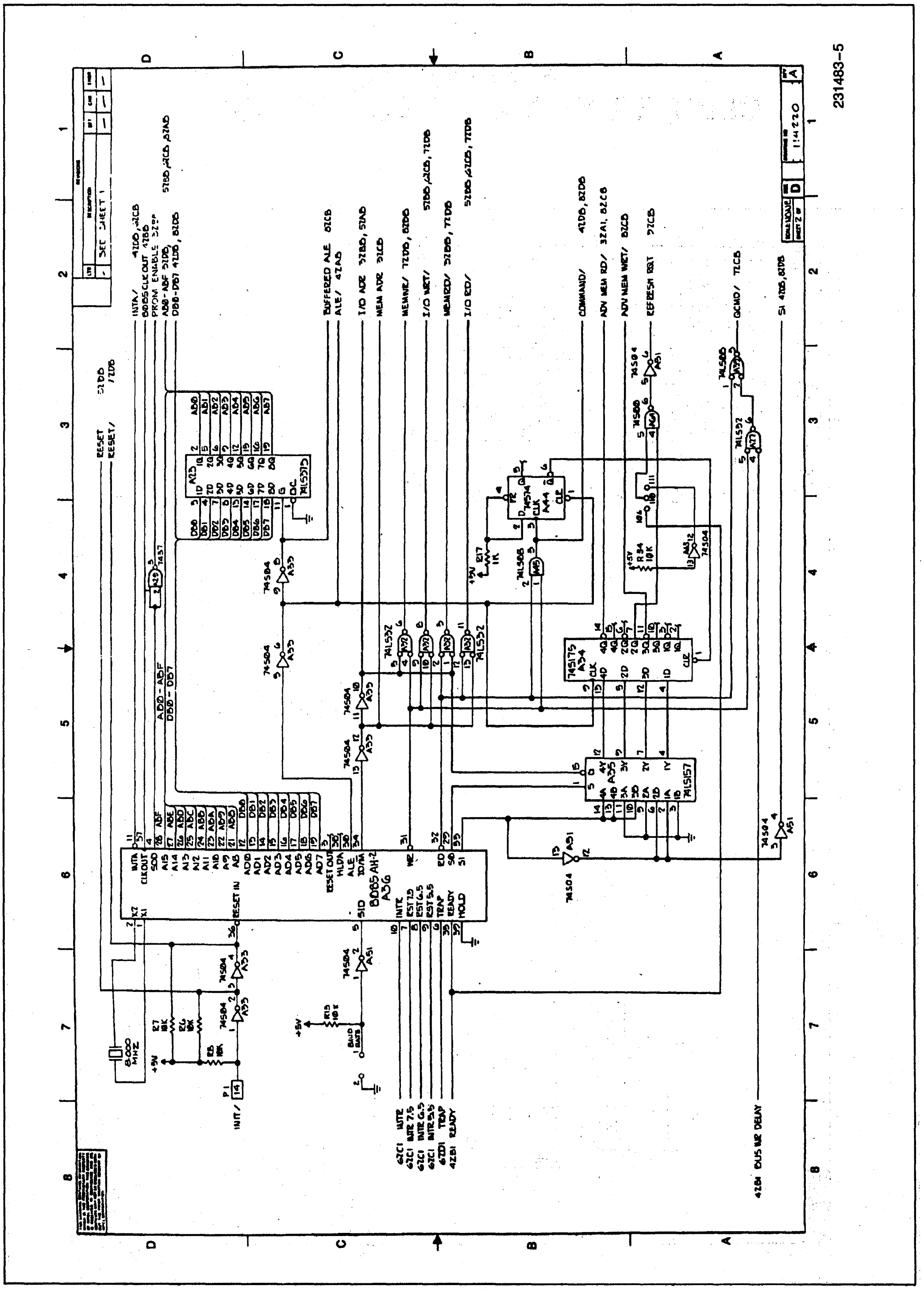

+12V
P1
78
C75
22 µF
15V
+5V
56
81 82
83 84
C19,21,29-32,36,38-44,46,49,50-52,55,60-63
.01 µF
25V
C53,99
.33 µF
50V
C73,74,77,99
22 µF
15V
C22-28,33,35,45,47,54,56,58,59,64-71,37,98
.1 µF
50V
GND
1 2
11 12
75 76
P2
C10
0.1 µF
50V
-12V
79 80
NOTES: UNLESS OTHERWISE SPECIFIED.
1. CAPACITANCE VALUES ARE IN MICROFARADS.
2. RESISTANCE VALUES ARE IN OHMS, 1/4W, 5%.
ECO 12-14384
ECO 12-14406
SCHEMATIC,
ISIS CLUSTER
NONE

DEVICE TABLE						
REF DES	DEVICE TYPE	POWER PINS				UNUSED LOGIC ELEMENT OUTPUT PINS
		GND	+5V			
A46,58,61,64,65	74S00	7	14			A65-3
A59	74LS02	7	14			A59-4
A26	74S03	7	14			A26-3,6,8
A33,40,51,63	74S04	7	14			A40-4,6,10,12; A63-10
A70,71	74LS04	7	14			A70-4,8
A41	74S08	7	14			
A39,45	74LS08	7	14			
A49	74S10	7	14			
A47	74S11	7	14			
A18	74H11	7	14			A18-6,12
A28	74S20	7	14			
A48	74LS27	7	14			A48-8
A42,62	74S32	7	14			
A27,32	74LS32	7	14			
A29	7437	7	14			A29-6,8
A44,55	74S74	7	14			A44-8,9
A31	74LS74	7	14			A31-8,9
A17,50	74LS138	8	16			
A56	74S140	7	14			
A35	74LS167	8	16			
A12	74163	8	16			
A34,43	74S175	8	16			A34-2,10,15; A43-2,3
A53,54	74LS240	10	20			
A76	74S373	10	20			
A23	74LS373	10	20			
A14	75188	7		14	1	
A15,16	75189A	7	14			A15-3,11; A16-6
A77-84	2164	8	16			
A68,69	8205	8	16			
A67	8218					
A36	8086AH-2	20	40			
A60	74367	8	16			
A66	8203-1	20	40			
A57	[illegible]	8	16			
A13	[illegible]	8	16			
A74,75	8226	8	16			
A22	8251A	4	26			
A21	8254	12	24			
A30	8259A	14	28			
A24,52,72,73	DP8304	10	20			
A38	9602	8	16			

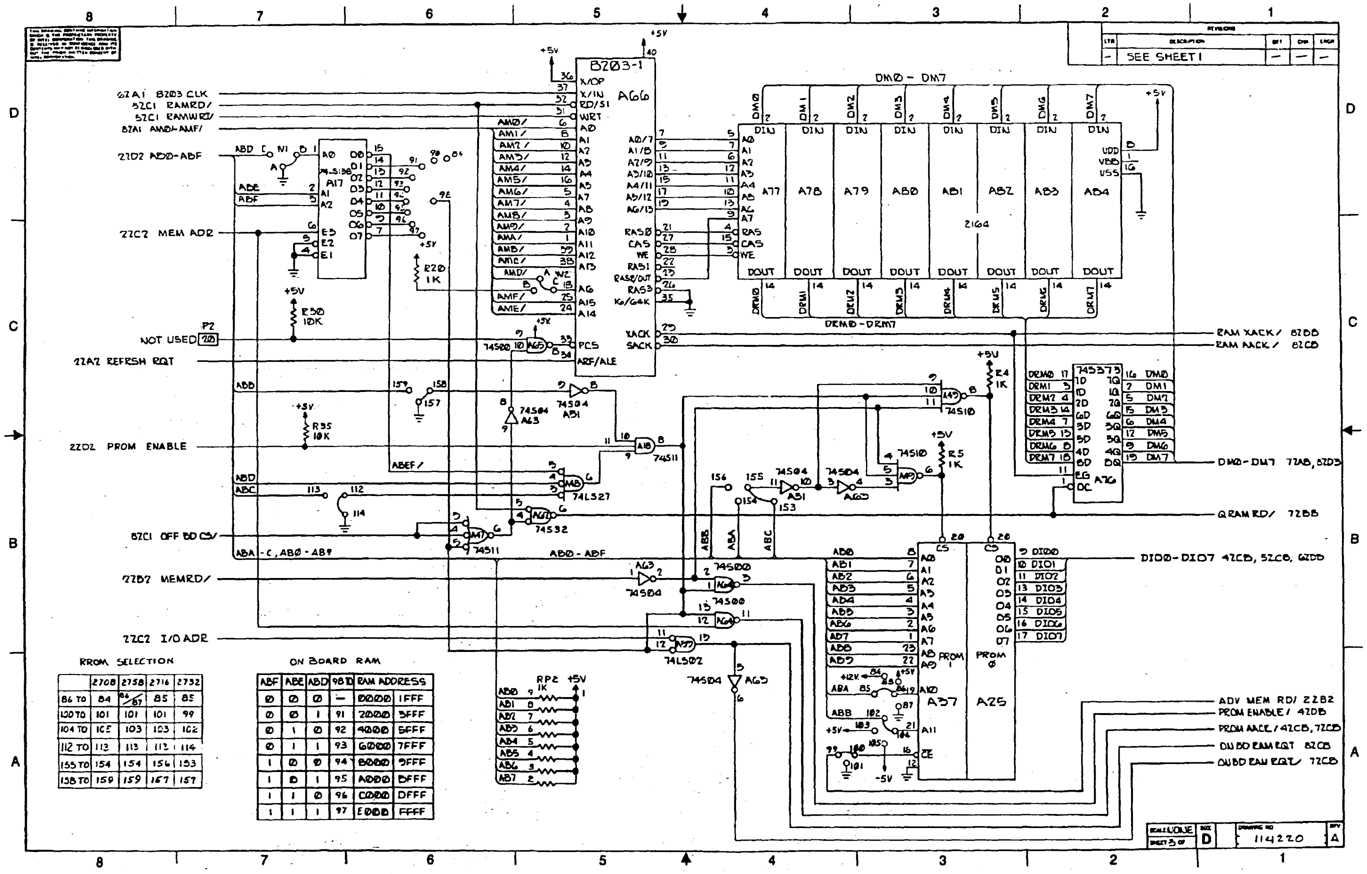
REFERENCE DESIGNATION	
LAST USED	NOT USED
A84	A1-11,19,20
C100	C20,34,72,76,78-92
	94,95
CR1	
E192	E3-45,48,71,74,78
	105,107,108,117,118
	134,135,144,171-180
	185
J3	J1,2
P2	
Q1	
R55	R3,9,33
RP2	RP1
W6	W4
Y2	

231483–4

NOTES: unless otherwise specified
1. Capacitance values are in microfarads.
2. Resistance values are in Ω, 1/4W, 5%.
3. Customer installed capacitor.


8085AH-2
A36
74LS373
A25
74LS175
A34
74LS157
A35
74LS74
A44
74S04
74LS32
74LS08
74LS02
8.000 MHZ
INTA/
8085CLKOUT
PROM ENABLE
AD0-AD7
DB0-DB7
BUFFERED ALE
ALE/
I/O ADR
MEM ADR
MEMWR/
I/O WRT/
MEMRD/
I/O RD/
COMMAND/
ADV MEM RD/
ADV MEM WRT/
REFRESH RQT
QCMD/
S1
RESET
RESET/
INIT/
INTR
INTR 7.5
INTR 6.5
INTR 5.5
TRAP
READY
DLS WR DELAY
BAUD RATE
+5V

231483-5



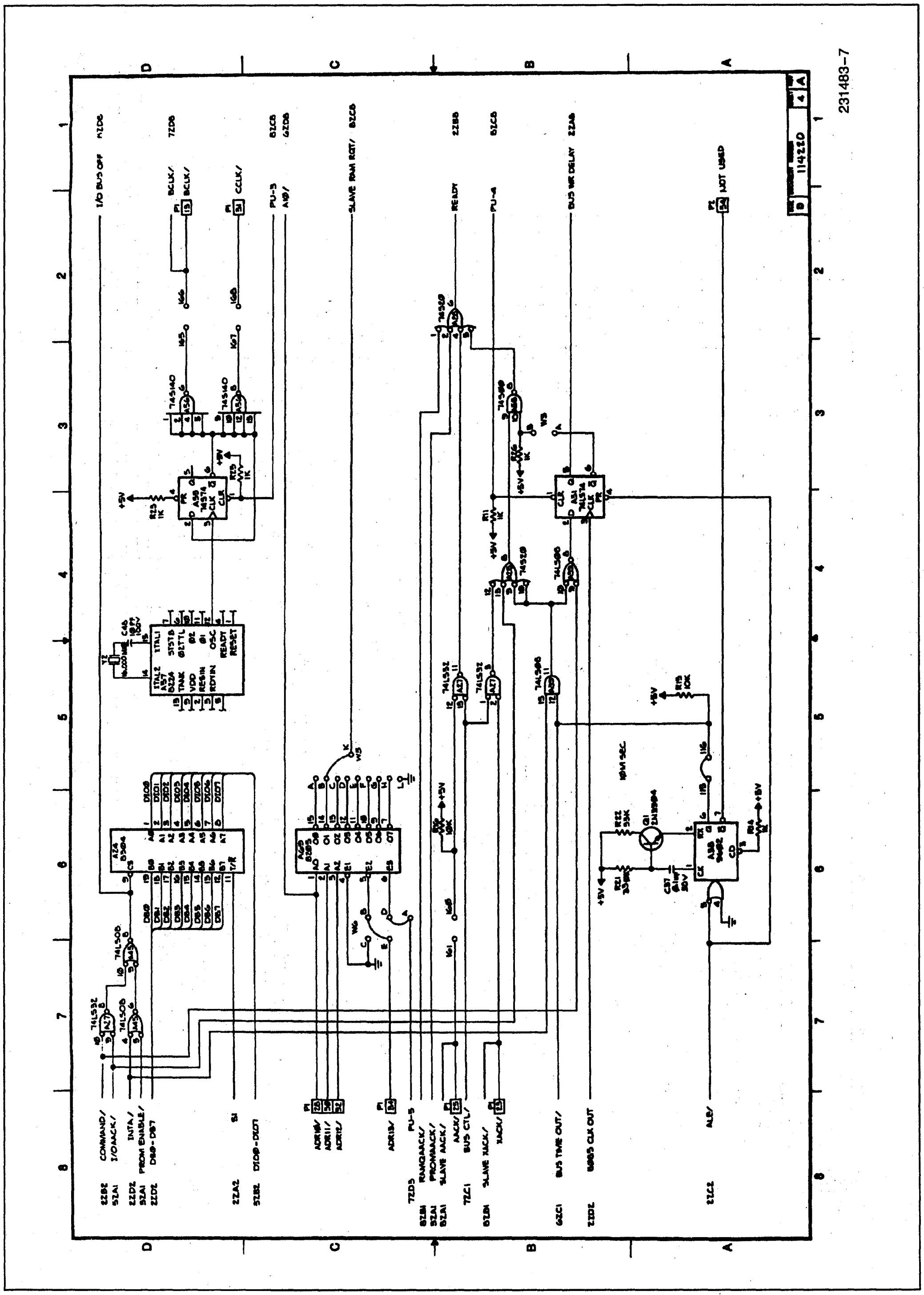
RROM SELECTION

	2708	2758	2716	2732
86 TO	84	86/87	85	85
100 TO	101	101	101	99
104 TO	105	103	103	102
112 TO	113	113	113	114
155 TO	154	154	156	153
158 TO	159	159	157	157

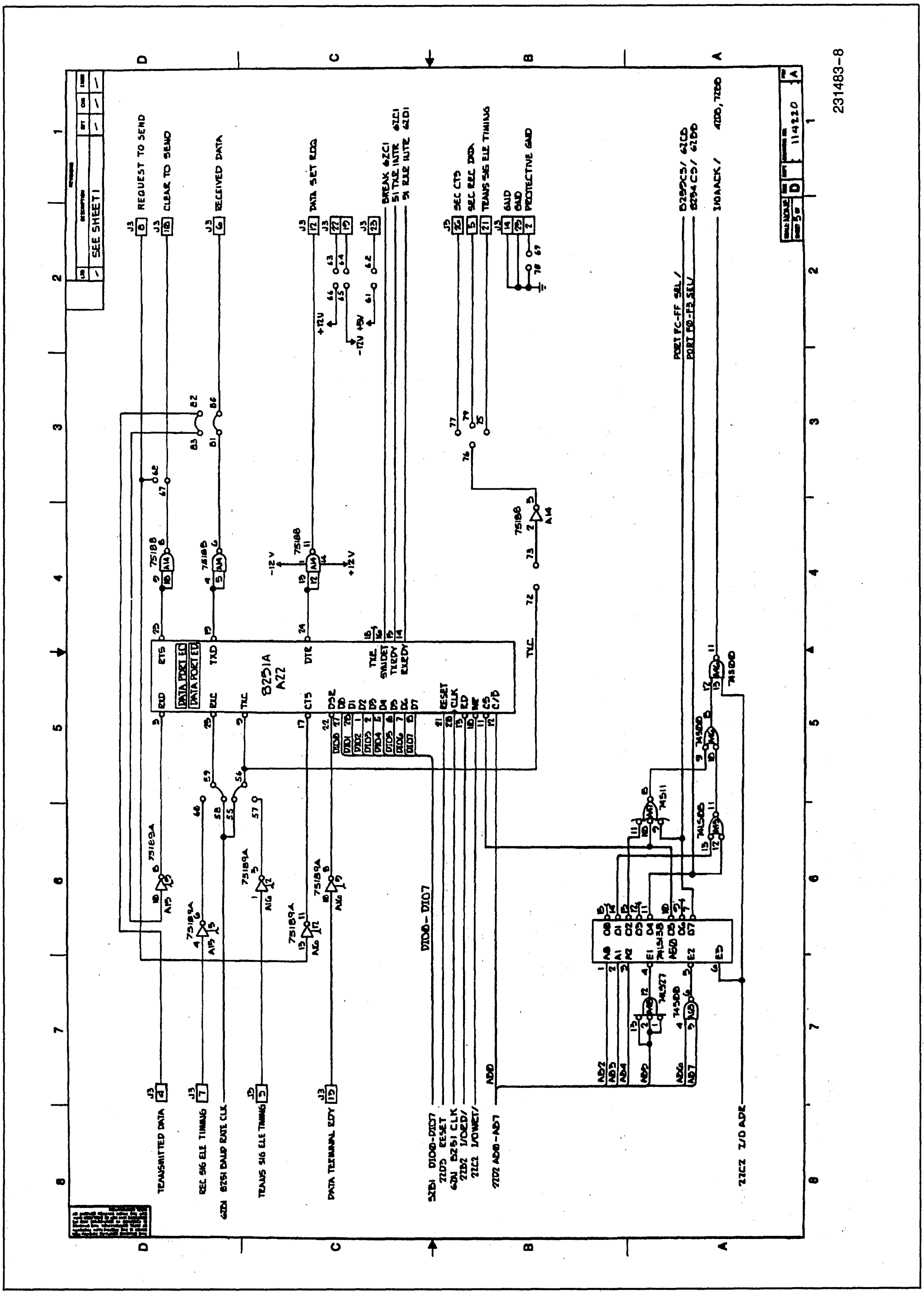
ON BOARD RAM

ABF	ABE	ABD	98 TO	RAM ADDRESS	
0	0	0	—	0000	1FFF
0	0	1	91	2000	3FFF
0	1	0	92	4000	5FFF
0	1	1	93	6000	7FFF
1	0	0	94	8000	9FFF
1	0	1	95	A000	BFFF
1	1	0	96	C000	DFFF
1	1	1	97	E000	FFFF

231483-6

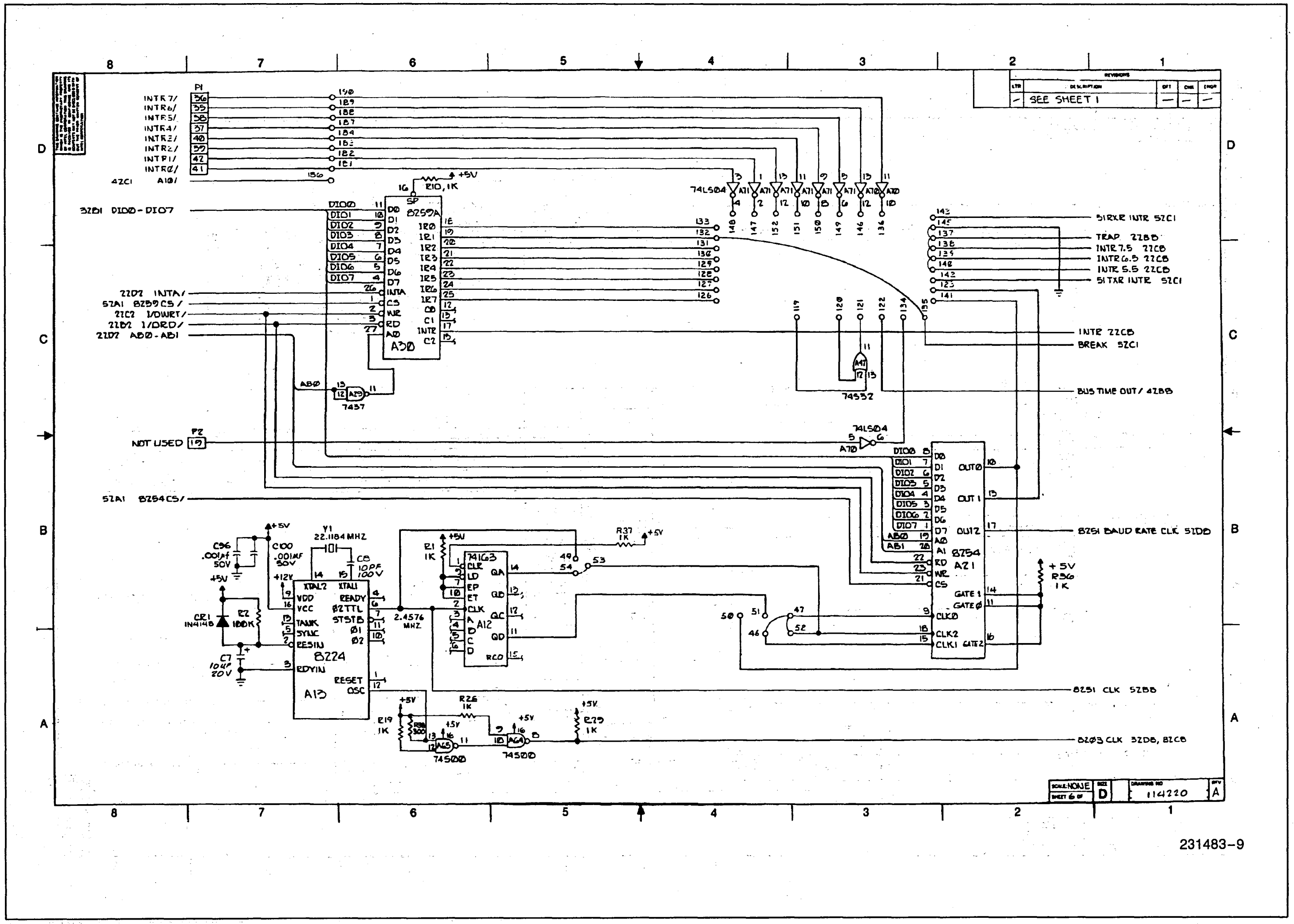

COMMAND/
I/OACK/
INTA/
PROM ENABLE/
DB0-DB7
SI
DIDØ-DID7
ADR10/
ADR11/
ADR12/
ADR13/
PU-5
RAMACK/
PROMACK/
SLAVE XACK/
XACK/
BUS CTL/
SLAVE XACK/
XACK/
BUS TIME OUT/
8085 CLK OUT
ALE/
I/O BUS OFF
BCLK/
CCLK/
PU-3
A19/
SLAVE RAM RQT/
READY
PU-4
BUS WR DELAY
NOT USED
74LS32
74LS08
74S140
74S74
74LS86
74LS74
8284
8205
3604
9602
2N3904
10M SEC
114220

231483-7



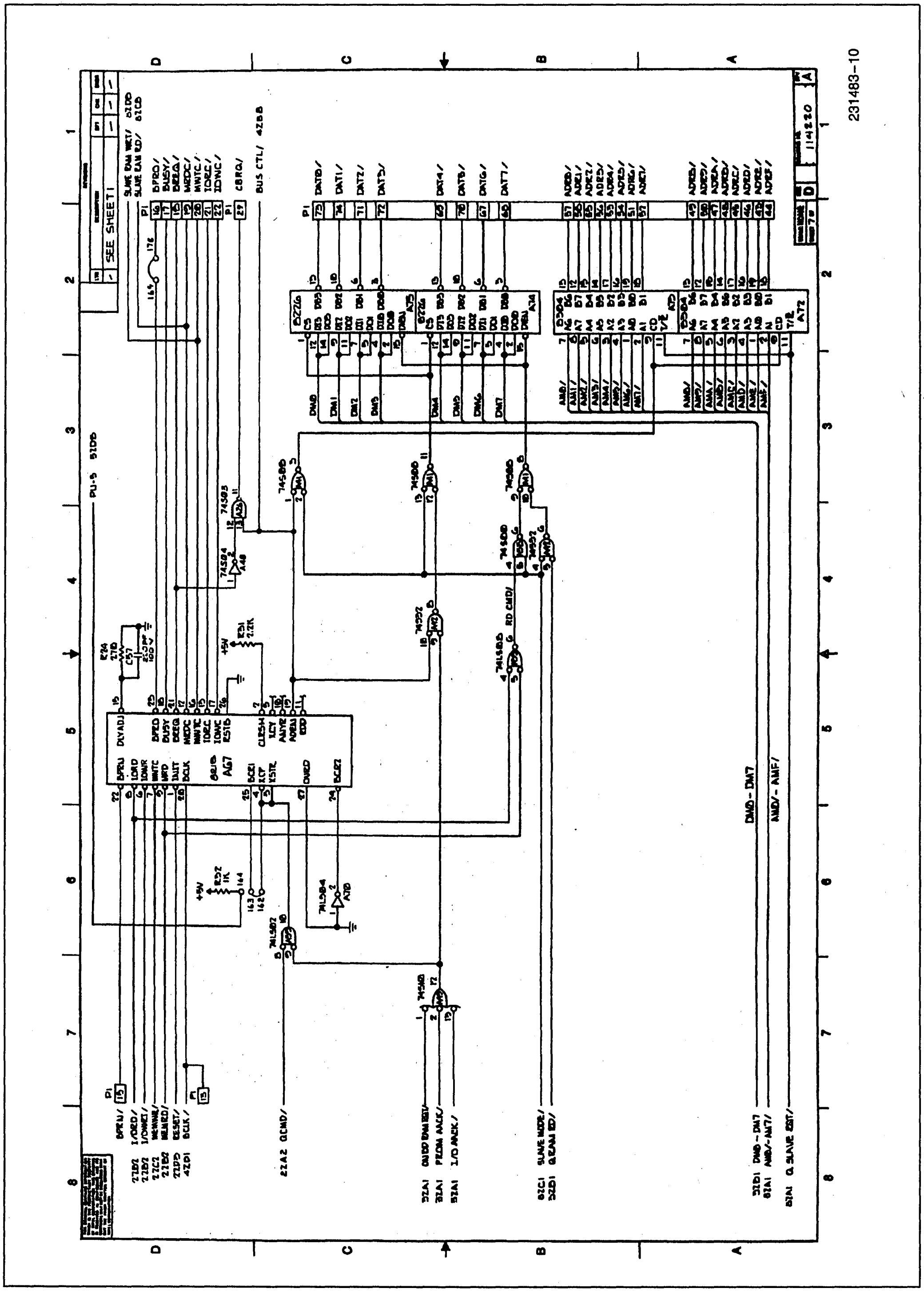
8251A
A22
DATA PORT E0
DATA PORT E1
RXD
TXD
RTS
DTR
CTS
DSR
TXC
RXC
SYNDET
TXRDY
RXRDY
RESET
CLK
RD
WR
CS
C/D
REQUEST TO SEND
CLEAR TO SEND
RECEIVED DATA
DATA SET RDY
BREAK 62C1
51 TXR INTR 62C1
51 RXR INTR 62D1
SEC CTS
SEC REC DATA
TRANS SIG ELE TIMING
GND
GND
PROTECTIVE GND
PORT FC-FF SEL/
PORT E0-E3 SEL/
8259CS/ 62C8
8254CS/ 62D0
IOACK/
TRANSMITTED DATA
REC SIG ELE TIMING
8251 BAUD RATE CLK
TRANS SIG ELE TIMING
DATA TERMINAL RDY
DIO0-DIO7
RESET
8251 CLK
I/ORD/
I/OWRT/
ADR0-ADR7
I/O ADR
75188
75189A
74LS138
74S00
74S11
74LS27
+12V
-12V
+5V
J3
J5
SEE SHEET 1
114220
D
A

231483-8

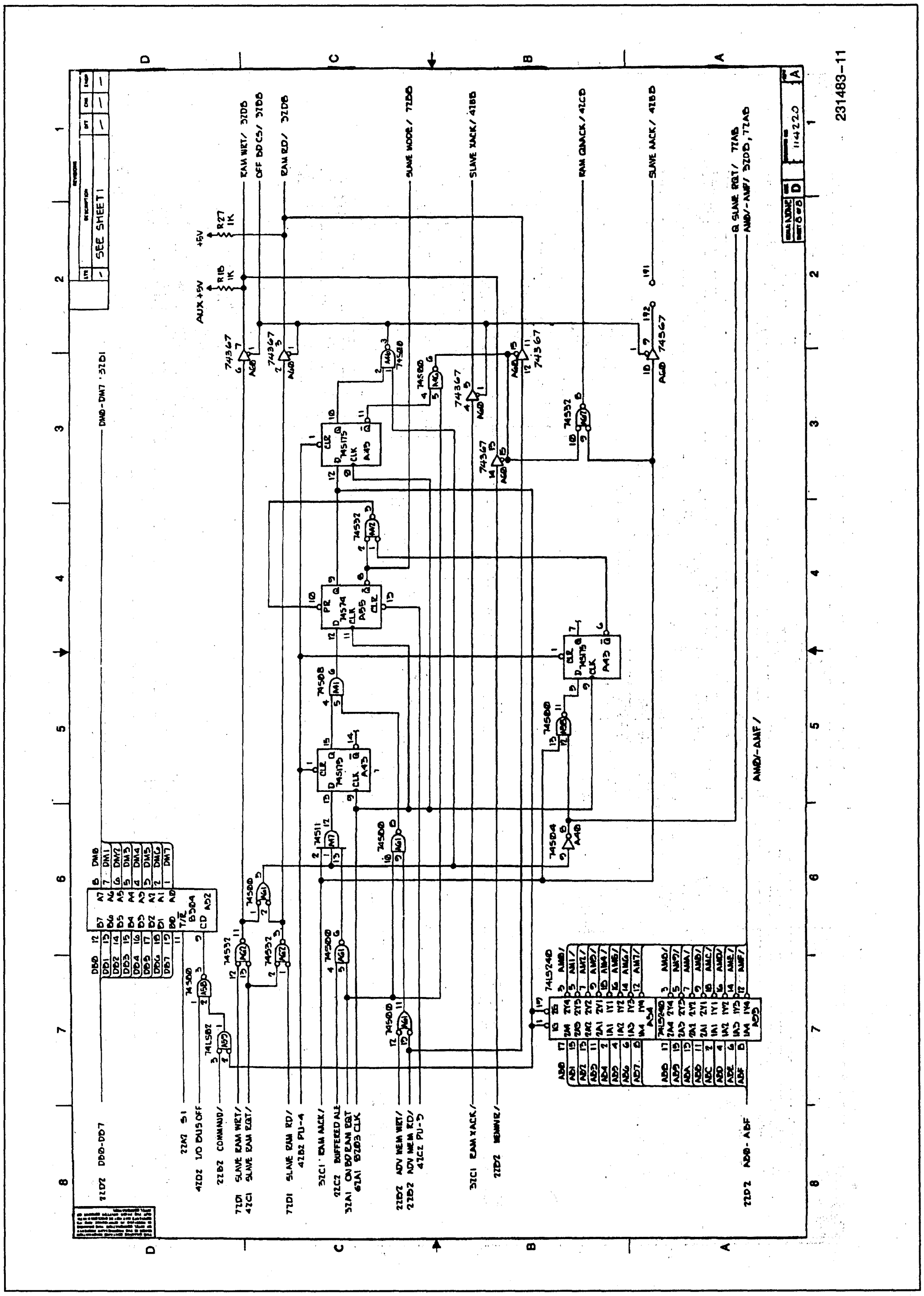


SEE SHEET 1
INTR7/ INTR6/ INTR5/ INTR4/ INTR3/ INTR2/ INTR1/ INTR0/
42C1 A10/
32B1 DIO0-DIO7
22D7 INTA/
52A1 8259CS/
22C2 I/OWRT/
22B2 I/ORD/
22D7 AB0-AB1
8259A A30
74LS04
74S32
7437
NOT USED
52A1 8254CS/
51RXR INTR 52C1
TRAP 22B8
INTR7.5 22C8
INTR6.5 22C8
INTR5.5 22C8
51TXR INTR 52C1
INTR 22C8
BREAK 52C1
BUS TIME OUT/ 42B8
8254 A21
8251 BAUD RATE CLK 51D8
8224 A13
22.1184 MHZ
2.4576 MHZ
74163 A12
8251 CLK 52B8
8203 CLK 32D8, 81C8
74S00
SCALE NONE
SHEET 6 OF
D
114220
A

231483-9


8216
8226
8304
74S00
74S02
74S04
74S32
74LS02
74LS04
DAT0/
DAT1/
DAT2/
DAT3/
DAT4/
DAT5/
DAT6/
DAT7/
BPRO/
BUSY/
BREQ/
MRDC/
MWTC/
IORC/
IOWC/
CBRQ/
BUS CTL/
SLAVE DMA WRT/
SLAVE DMA RD/
RD CMD/
DMD - DM7
AMD/ - AMF/
PU-5
BPRN/
I/ORD/
I/OWRT/
MEMWR/
MEMRD/
RESET/
BCLK/
QCMD/
SLAVE MODE/
Q.DMA RD/
Q.SLAVE RST/
I/O ACK/
PROM ACK/
SEE SHEET 1
+5V
R51 2.2K
R52 1K
R34 270
C57 220pF 100V

231483-10


SEE SHEET 1
DMD-DM7 / 37D1
DD0-DD7
ADD-ADF
AMD/-AMF/
RAM WRT/ 37D8
OFF DDCS/ 37D8
RAM RD/ 37D8
SLAVE MODE/ 72D8
SLAVE XACK/ 41B8
RAM GNACK/ 42C8
SLAVE AACK/ 41B8
G SLAVE RQT/ 72A8
AMD/-AMF/ 32D8, 72A8
AUX +5V
+5V
R18 1K
R27 1K
74367
74S00
74S32
74S08
74S11
74S04
74LS02
74S175
74S74
74S240
8304
1/O BUS OFF
COMMAND/
SLAVE RAM WRT/
SLAVE RAM RQT/
SLAVE RAM RD/
RAM AACK/
BUFFERED ALE
ON BD RAM RQT
8203 CLK
ADV MEM WRT/
ADV MEM RD/
RAM XACK/
MEMWR/

231483-11

intel®

APPLICATION NOTE

AP-278

October 1986

Integrating the PC AT Into the Intel Development Environment

SRIVATS SAMPATH
DSO APPLICATIONS

Order Number: 280272-001

INTRODUCTION

In recent years the Personal Computer has become a popular vehicle for delivering computing power to the engineers. IBM's latest offering the Personal Computer AT (Advanced Technology) incorporating INTEL's 80286 16-bit microprocessor, brings about a high level of technical sophistication into a personal computer. The power, speed and memory addressability of the 80286 microprocessor is now available to the user to do tasks which at one time could be run only on a mini or mainframe.

Intel has recognized this growing trend and has introduced translators, debuggers and networking for the PC AT. The same tools that have in the past, been offered only on Intel's proprietary development systems are now available on the PC AT under PC-DOS 3.0 or greater. Language translators are available for the complete spectrum of Intel microcontrollers (MCS®-51 and MCS-96 families) and microprocessors (8086, 80186, 80286). 80386 tools will be introduced early 1987. This is the first time powerful software debuggers like PSCOPE and TSCOPE, hardware debuggers like I^{2}ICE™ have been made available on a personal computer. Intel already supports a broad range of workstations which may be networked to form a productive network. This application note discusses the multiple methods in which the PC AT may be integrated into this development environment.

All software discussed in this application note is available (except where listed) in the Network Toolbox, Part No NDS2TLB 2.0, as discussed in Appendix D.

THE INTEL DEVELOPMENT ENVIRONMENT

The Intel development environment consists of iPDS™ Personal Development System, Series-II's, Series-III's, Series-IV's all of which can be operated standalone or networked through a file sever using the NRM (Network Resource Manager). Intel also supports industry standard hosts such as the DEC VAX and now we include the PC AT as a supported host. Figure 1 shows all combinations of the Intel development environment while Figure 2 illustrates the possible inclusion points of the personal computer.

This appliction note assumes that the reader is familiar with the current Intel Development Environmanet. For more information on the Intel Development Envionment please refer to:

1. AP Note Number AP-244 *DJC A Key To Increased Network Productivity*
2. AP Note Number AP-245 *Creating an Efficient HFS*

These application notes are also available in the 1986 DSO Handbook, Order Number 210940.

This application note deals with integrating the PC AT into an existing Intel development environment. The enclosed details will allow the reader to choose the method that best suits the project. This applications note will discuss three methods of data transfer—serial interconnect, media transfer and networking. Of these, serial transfer is the most universal, media transfer is the most straightforward and Ethernet transfer is the most efficient.

SERIAL INTERCONNECTS

The simplest method of integrating the PC AT into the Intel development environment is through serial interconnects. This method is inexpensive albeit slow. Serial interconnects also allows the user the flexibility to use modems to interconnect with systems which are not in the same location. It allows for both terminal emulation and file transfer at speeds of up to 9600 baud. Serial connections have always been a popular method of linking different computers. Unfortunately this has resulted in a variety of serial communication software being developed that are incompatible over different operating systems. Intel recognized this incompatibility and decided to advocate serial communication software that was compatible over a range of operating systems. The KERMIT file transfer protocol developed by Columbia University, addressed all the needs of serial interconnects, and resolved most inadequacies in previously available serial software. It also solved the multiple operating system incompatibility issue. KERMIT, is available for all hosts shown in Figure 1. Note however that all KERMIT implementations are not equal. The specification details a minimal set of and also specifies numerous additional features which may be added. Currently, the ISIS implementation on the iPDS system, Series II, III and Series IV is a minimal set while the VAX and PC implementations are both extensive. The XENIX and iRMX versions are good and being improved. KERMIT is public domain software and cannot be charged for. Versions for the Intel hosts are available from Insite for a small disk copying fee (see Appendix C).

The following paragraphs discuss the different methods of integrating the PC AT using the KERMIT file transfer protocol. Intel, with help from a number of customers presently using our systems, has developed KERMIT software for the following systems:

- iPDS—Serial port
- Series-II or III—Serial port
- Series-IV—Serial port 2
- ISIS cluster board—Directly into the ISIS cluster board

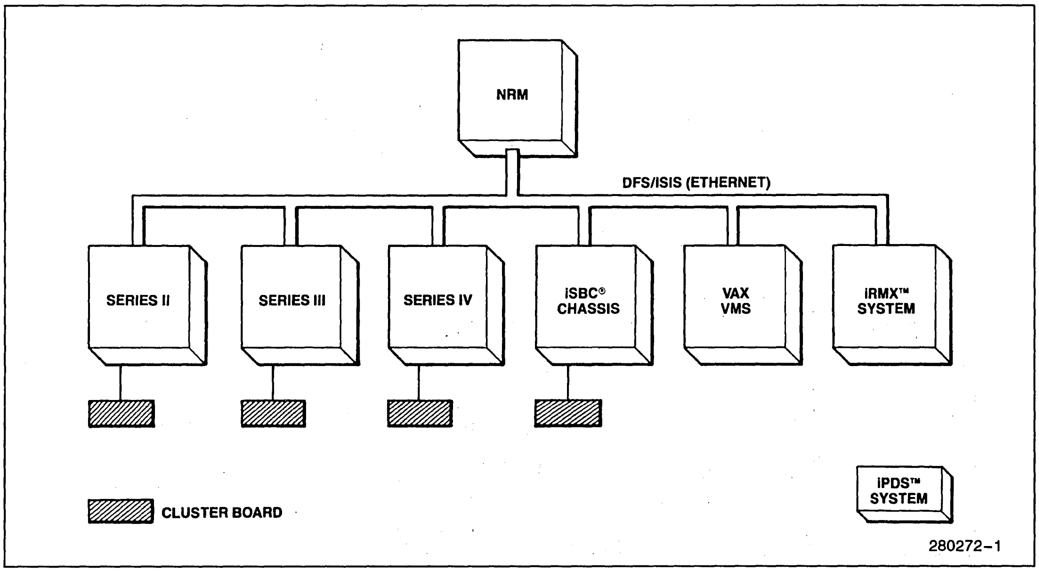


Figure 1. The Intel Development Environment

KERMIT

KERMIT is a file transfer and terminal emulation protocol developed by Columbia University in 1981. Since then KERMIT has been ported to over 30 different systems and is on its way in becoming an industry standard protocol. The KERMIT protocol is designed around character oriented transmission over serial lines. The design allows for peculiarities in transmission medium and requirements of different operating environments. The KERMIT protocol incorporated features and ideas from protocols like DIALNET, DECNET and APPANET. Currently KERMIT has been implemented in over 26 systems. A detailed discussion on the KERMIT protocol is covered in Appendix A.

The following list shows the different systems and operating systems that support the KERMIT protocol.

System	O/S
Series-II, III, IV, iPDS	ISIS
IBM 370 Series	VM/CMS, MVS/TSO, MTS
CDC Cyber	NOS
DEC VAX-11/7XX	VSM, UNIX
PC	MS-DOS, PC-DOS
Apollo	Aegis
PRIME	PRIMOS
HP 3000, 1000	
Apple 11 6502	Apple DOS

KERMIT is a two ended protocol. It needs the remote system to have KERMIT running on it too, to do file transfers. The KERMIT executing on the PC is MS-KERMIT and the one on the Series-II, III, IV, iPDS is the ISIS-KERMIT. The following chapters will detail how this serial interconnect is established.

KERMIT can communicate over either port on the PC AT. Switching between the PC ports can help the PC user communicate with two different systems alternatively.

MS-KERMIT

MS-KERMIT is a program that implements the KERMIT file transfer protocol for the IBM PC AT and several other machines using the same processor family (Intel 8088 or 8086) and operating systems family (PC-DOS or MS-DOS 2.0 or greater).

MS-KERMIT has an extensive command set. A brief summary is shown in Figure 4 wih a more detailed explanation in Appendix A.

ISIS KERMIT

ISIS KERMIT is a minimal KERMIT implementation. This is also available in Insite as described in Appendix C.

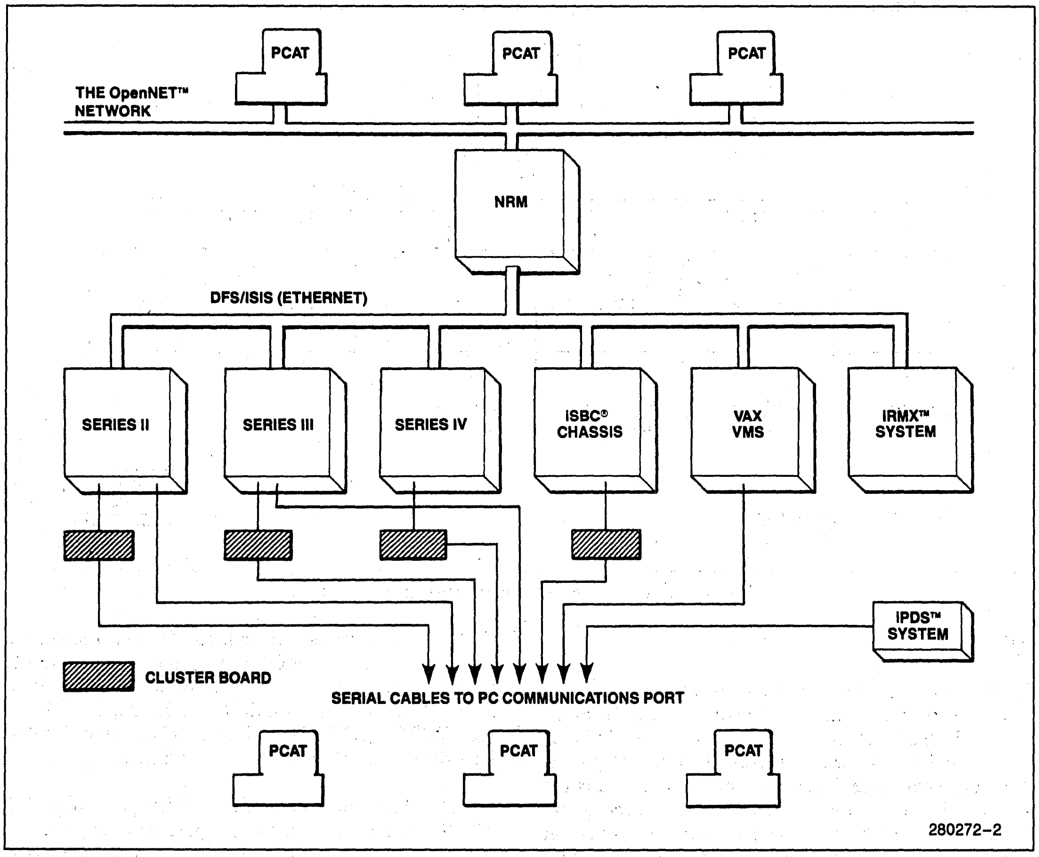


Figure 2. Including the PC

It operates under the ISIS operating system. The basic command set supported by ISIS KERMIT are:

CONNECT—enters terminal mode for communication with host.

DEBUG—toggles debug mode on/off. Prints messages during transfers. Normally used only during troubleshooting.

EXIT—Return back to ISIS.

SEND filename—specifies the file to be transferred to host. May use the ISIS :fn: drive designation to open file on any logical drive in the system. That drive designator will be stripped from the name before it is sent to the host.

RECEIVE [n]—After commanding the host to send a particular file, or a group of files (wildcards can be used on hosts if they're smart enough), press 'HOME' or 'control]' to drop back to ISIS-KERMIT and enter the 'RECEIVE' command. If the command is followed by a number (0–9) the files(s) will be sent to that logical drive. For example, 'REC 4' will cause the filename(s) to be prefixed by :f4: when opened. The number of drives varies, depending on which system is used.

Since there are only 5 commands, a single letter is all that is required to use them.

'r 3' is euivalent to 'RECEIVE 3'

KERMIT is invoked as follows:

KERMIT [baud-rate] [port number]

The default baud rate is 2400, Others available are 300, 1200, 9600.

The port number selection is effective only on Series II. The iPDS system has only one port, and the Series IV must use to port 2, since it is global in multi-user mode.

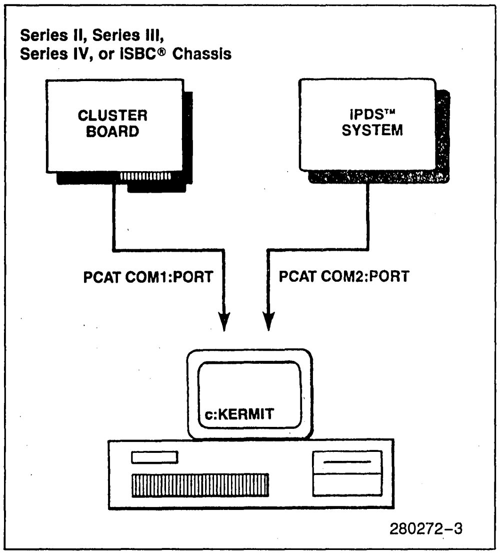


Figure 3. How to Connect the PC to a Remote Host Using KERMIT

A TYPICAL KERMIT SESSION

With the availability of 8051 and 8044 languages on DOS, an existing user may need to move existing software from the iPDS system to the PC AT. The following paragraphs serve both as an example, and as a method to help the iPDS user set up his serial interconnects to do the migration. The various steps, which are explained in detail, include setting up the iPDS system for serial communications, setting up the PC and the actual terminal emulation and file transfer sequence. The steps illustrate the ease with which this migration is brought about.

Command	Explanation
CONNECT	To connect as a remote terminal to a remote system.
DELETE	Delete local files
LOCAL	Prefix for local file management commands
RECEIVE	Receive files from remote system
SEND	Send files to remote system
QUIT	Quit from MS-KERMIT
RUN	Execute a MS-DOS program
SET	Set parameters like baud rate, serial channel
SHOW	Display all parameters
EXI	Exit from MS-KERMIT
DIRECTORY	Directory of local PC

Figure 4. KERMIT Command Set

STEP 1.

Install ISIS KERMIT on a iPDS diskette, and create a CSD file ABOOT.CSD that looks like this:

```
SERIAL A    B = 9600
  ;Set the serial port in ASYNC mode at 9600
ASSIGN :CO:TO :SO:
  ;Redirect console out to the serial port
ASSIGN :CI:TO :SI:
  ;Redirect console in to the serial port
```

This step sets up the iPDS for serial communication by initializing the serial port to communicate asynchronously at 9600 baud. The console I/O redirection is done to enable KERMIT-MS to control the iPDS. Placing these commands in the ABOOT.CSD file help bring up the iPDS system in the right mode whenever it is reset.

STEP 2.

Install DOS KERMIT on the PC AT and invoke it by typing KERMIT from the command line.

```
C: \> KERMIT

IBM-PC KERMIT-MS VER 2.26
TYPE? FOR HELP
KEMIT-MS > SET BAUD 9600
KERMIT-MS >connect
```

Once a successful connection has been made to the iPDS the PC AT terminal will display the iPDS prompt.

```
A0>
```

Now the user can do any operation like DIR, ASSIGN, etc., on the iPDS system from the PC keyboard.

STEP 3.

```
FOR FILE TRANSFER.
```

Invoke the iPDS KERMIT by typing in KERMIT

```
A0>KERMIT 9600 1 ;9600= baud rate and
1=port
```

The ISIS KERMIT prompt will appear ISIS-KERMIT>

For receiving files type in

```
ISIS-KERMIT>RECEIVE :F0:
```

Exit back to the PC by typing in CNTRL] C at the same time. The user is now back to the KERMIT-MS> prompt. Now type in

```
KERMIT-MS>SEND EXAMPLE.BAT
```

A Screen comes up showing data transfer status and on successful completion on file transfer will give back the prompt. More information on setting the number of retries on error packets and timeouts are explained in Appendix A.

```
KERMIT-MS>
```

MEDIA TRANSFER UTILITIES

Diskettes constitute the main source for data and information storage. Most software is kept on diskettes for ease of storage, transportability, and safekeeping. Migrating from one host system to another involves transferring this data on to the new host system media. Media transfer is useful only if both hosts are at the same site and support a compatible peripheral device. If both these conditions are met, media transfer provides a fast convenient way for casual data transfer.

Handling diskettes and interacting with two host computer systems is error-prone and inconvenient. I recommended the other two methods of file transfer in a production environment where the two computers may converse without operator intervention. One major problem with media transfer is the lack of industry standards. There is an 8 inch single density standard (IBM 3740) but not for other densities nor for 5¼ inch media. To solve this problem, special host dependent utility programs must be written to permit the reading of another systems diskettes. This was addressed for the PC by developing a set of utilities that allow file transfer from 5¼ inch and 8 inch. This gives the user the ability to move between the Series-IV environment and the PC-DOS environment with the least overhead and loss of productivity. A key factor in projects these days.

MSCOPY is program that manipulates a MS-DOC disk on a Series IV or NRM. It also helps the PC AT user access the NRM print spooler. The user can copy software both to and from a Series-IV. MSCOPY expects the MS-DOS diskette in FL0. While running MSCOPY do NOT change the disk as MSCOPY keeps the Disk allocation table in memory and will not reread them from a new disk but will write out the old table and directory.

MSCOPY supports 48 or 96 TPI disks, 8 or 9 sectors per track, 1 or 2 heads, MS-DOS vers. 1, 2 or 3. It does not support 1.2 Mb high density diskettes.

It has two modes of operation, interactive and non-interactive. In the non-interactive mode you may enter only one command and it must deal only with the MS-DOS root directory. To use the non-interactive type the command on the invocation line.

In the interactive mode, (entered by invoking MSCOPY with no parameters) MSCOPY will prompt you for a command. Currently there are seven legal commands.

The seven commands are:

READ msfile indxfile—Copies msfile to indxfile. msfile must be in the current directory. indxfile can be any valid iNDX pathname up to 40 characters long.

WRITE indxfile msfile—Copies indxfile to msfile. msfile will be added to the current directory. indxfile can be any valid iNDX pathname up to 40 characters long.

CD msdir—Changes the current directory to the directory msdir. This command will only go up or down the tree one node at a time. To go back one level say "CD". To go deeper say "CD name" where name is a dir entry in the current directory. Typing in "CD \" will jump to the root directory.

DELETE msfile—Removes msfile from the current directory and reclaims the space it occupied.

RELAB label name—Will change or add the volume name of the MS-DOS disk. Label name may be up to eleven characters long.

DIR—Will display the current directory.

EXIT—To return to iNDX.

8″ ISIS Media

Flagstaff Engineering in Phoenix, Arizona have a set of tools that allow direct transfer from 8″ ISIS (SS/SD) media to PC's. The tool set consists of a add-on board for the PC, an 8″ drive and the driver software to do the required transfer. File transfer is bidirectional. The program ISS8TO5 copies files from 8 inch media to PC, and ISS5TO8 copies the other way. An example is shown in Figure 5.

Please note that Intel does not sell, support or warrant reliability of this product. Intel's evaluation sample has proved reliable and Flagstaff technical support has been good.

For more information please contact:

Flagstaff Engineering
Box 1970
Flagstaff, AZ 86001

```
>ISS8T05
COPY INTEL ISIS DISKETTE FILE TO IBM PC-DOS FILE PROGRAM
COPYRIGHT FLAGSTAFF ENGINEERING 10/17/83

THIS PROGRAM WILL COPY A FILE FROM A 8" ISIS SINGLE DENSITY DISKETTE TO AN IBM
PC-DOS DATA FILE. THE FILES MUST BE CREATED USING ISIS-II OR RMX/80 SYSTEMS.

INSERT 8" ISIS DISKETTE--ENTER DRIVE(1/2) WHEN READY.?
FILE DIRECTORY FOR DISKETTE 164539001
01-ISIS    .DIR(025)  02-ISIS    .MAP(002)  03-ISIS    .TO(023)
04-ICE51   .   (253)  05-ICE51   .OV0(020)  06-ICE51   .OV1(011)
07-ICE51   .OV3(027)  08-ICE51   .OV4(008)  09-ICE51   .OV5(036)
10-ICE51   .OVE(082)  11-ICE51   .OVH(498)  12-ICE51   .OVS(049)

ENTER ISIS FILE NUMBER (1-96/99=ALL)--PRESS ENTER IF NONE?
DO YOU WANT TO COPY FROM ANOTHER ISIS DISKETTE (N/Y)?
```

Figure 5

NETWORKING THE PC AT WITH THE OpenNET™ SYSTEM

OpenNET™ Architecture

OpenNET is Intel's Local Area Network architecture. OpenNET conforms to the Open Systems Interconnect (OSI) model defined by the International Standards Organization (ISO). The major objective of ISO is to create an open systems networking environment where any vendor's computer system can be connected to any network and freely share data with the network.

The OSI ISO architecture is based on a seven layer model (see Figure 6). The seven layers isolate independent functions so that the network can better make use of new software and hardware without adversely affecting the other layers. The upper three layers (5 through 7) provide interoperation functions, while the lower four layers (1 through 4) provide interconnect functions and the bottom two layers (1 through 2) are concerned with the transmission through physical medium.

OpenNET™ Family

As part of Intel's Open Development Environment (ODE), OpenNET supports a number of industry standard hosts and operating systems. To date, OpenNET runs on the IBM PC family with PC-DOS 3.1 or greater, iRMX, XENIX and iNDX as shown in Figure 7.

Since all of the above mentioned systems conform to the ISO seven layer model, they can all interoperate and interconnect over the same network.

OpenNET™ PC Link: Overview

Intel's PC connection on the OpenNET system, named OpenNET PC Link, consists of an add-in controller board for the PC, XT or AT (Layers 1–2), the iNA960 ISO transport software (layers 3–4) and the MS-NET software (layers 5–7).

PC AND THE NRM FILE SERVER ON THE OpenNET™ SYSTEM

The remainder of this application note discusses the use of a PC on the OpenNET system with Intel's Nework Resource Manager (NRM) as the OpenNET file server.

The current NDS-II NRM can be converted into an OpenNET file server by installing the iSXM™ 552 board and iNDX R3.0 or greater software (the board and the software have been kitted into the "NDS-II OpenNET Upgrade Kit", Part # "iMDX555"). New users have the choice of a Mini OpenNET NRM with a 40 Mb disk or a Maxi OpenNET NRM with a 140 Mb disk (upgradeable to 4 140 Mb disks) and a 60 Mb tape.

DETAILED EXPLANATION OF CONNECTING PC TO THE OpenNET™ SYSTEM

The OpenNET system uses concepts such as SERVERS and CONSUMERS which allow a building block approach to creating a network that can be tailored to your particular specification. The following chapters will discuss indepth the various concepts of the OpenNET system and on how to implement PC's and iNDX systems on an OpenNET network.

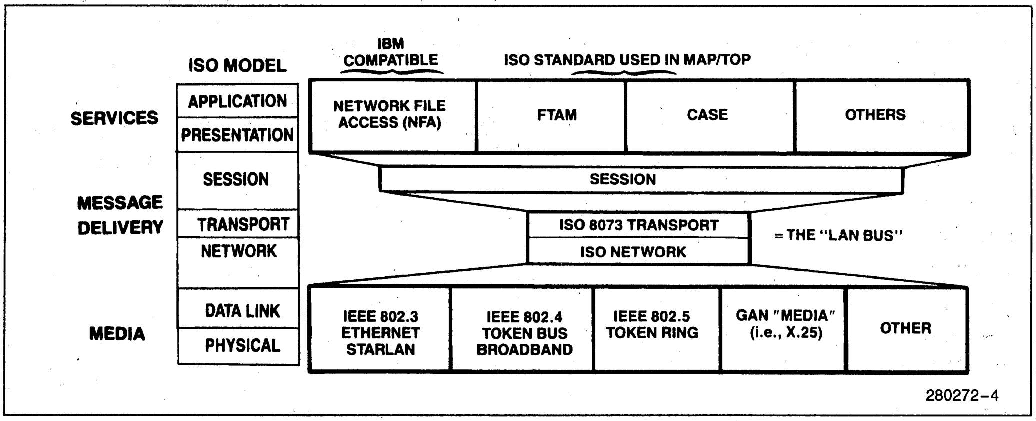


Figure 6

A server is defined as a system on which network resources like files, directories and a printer are kept. A Server usually has a number of hard disks. It is called a "SERVER" because it serves the other systems on the network when they request for files and printer service. There may be a number of servers on the network. The NRM with the iSXM 552 board and iNDX 3.0 installed in it acts as a SERVER for the other systems on the OpenNET network. XENIX and iRMX system can also be servers.

Computers that are linked to a server and use it as a resource for files and printer service are called CONSUMERS. CONSUMERS can also operate independent of a SERVER. An example of a CONSUMER on the OpenNET network is the PC. It can operate independently as a workstation and also uses the NRM for file services. XENIX and iRMX are capable of operating as consumers too.

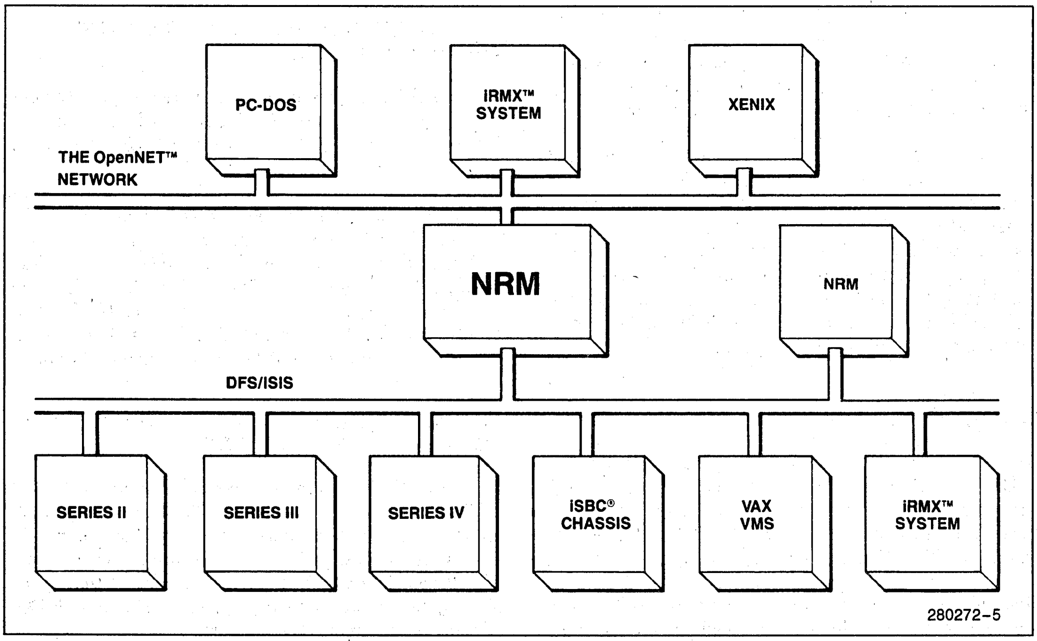


Figure 7

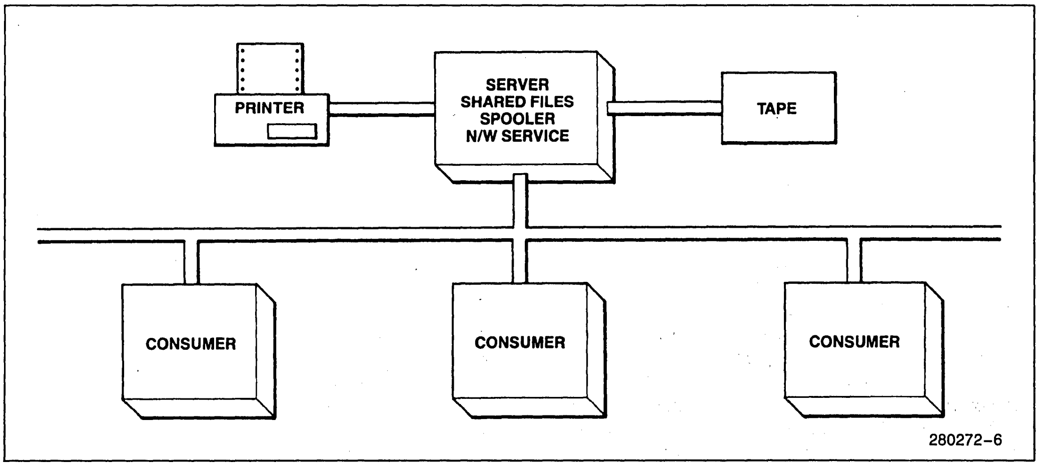


Figure 8. Server and Consumer

A list of all servers and consumers on an OpenNET system is stored in a database file called the NETADDR file. This is discussed in Appendix C.

Figure 8 illustrates how the server and consumers interact on a network.

A SAMPLE OpenNET™ SESSION FROM THE PC TO THE NRM

The following paragraphs will describe how the PC user can access files at the NRM. But before going into the details a few hints on making the process automatic and easy.

Since the OpenNET system uses the concept of virtual drives, it will be beneficial to have as many virtual drives as possible. Refer to the Virtual Drives section under the Chapter "Connecting PC's to OpenNET". DOS 3.1 has a default number of virtual drives 5 (A: through E:), however for OpenNET more drives may be needed. This can be achieved through modifying the CONFIG.SYS file in the root directory of the PC. Edit this file to include the command:

```
lastdrive = z  ;increase the number of
                virtual drives to 26.
```

A typical CONFIG.SYS file is shown in Figure 9.

On boot up, DOS 3.1 will read this file and automatically configure the PC as specified.

Now start the PC as a consumer by entering:

```
    C:NET START RDR <this PC name>
```

This is specified in the NETADDR file discussed in Appendix C.

This command loads the PC-LINK communication software onto the PC-Link board, and sets up the environment for communicating with any server. This sets up the session layer on the controller board.

If all the steps went through successfully, the network software will be loaded into the PC-Link board and will sign on with:

```
*******************************************
*                                         *
*             OpenNET™ PC Link            *
*                                         *
*   Copyright 1985, Intel Corporation     *
*                                         *
*******************************************

C:>
```

Now connect to the NRM using the NET Use command. The syntax for this command is:

```
C:> NET USE <virtual drive>\\<server name>
 \username password
```

Example:

```
C:> NET USE J:\\APPS--NRM1/GUEST WELCOME
```

The above command creates a virtual drive J: which points to the home directory of the user GUEST with a password WELCOME at APPS—NRM1. This drive J: is like any PC drive except that it points to a directory

```
lastdrive = z           ;Set the number of virtual drives to 26
device = \sys\ansi.sys  ;Set the terminal characteristics to ANSI
files = 20              ;Number of files open at one single time
buffers = 20            ;Number of buffers for file I/O
break on                ;set break key on
```

Figure 9

at a remote NRM. The user can do any DOS function like copy, dir, etc. even execute DOS applications programs that are stored at the NRM in this directory. Just by having this capacity the PC becomes a very powerful and flexible workstation. By sitting at one PC the user can have access to several file servers.

The command will come back with a message "command successfully completed" if it was successful, otherwise it will wait about 4 minutes before timing out and giving back the DOS prompt.

The user can now use this virtual drive just as if it was any other PC drive. For example a DIR command on drive J: will look like this.

```
C:>dir J:

Volume in drive J has no label
Directory of J:/

INIT      BAK        83   9-06-85   9:23a
INIT      CSD       290   9-06-85   9:23A
CDISK     CPM    401408   9-06-85   9:24a
NNMAC     MAC        74   9-06-85   9:24A
HILIB           <DIR>     9-06-85   9:26a
DJC             <DIR>     9-06-85   9:26a
DATABASE  DIR   <DIR>     9-06-85   9:55a
KERMIT    DIR   <DIR>     9-06-85   9:57a
IMPORT    DIR   <DIR>     9-06-85  10:15a
OPENNET   DIR   <DIR>     9-06-85  10:28a
     10 File(s) 30642176 bytes free
```

This is the home directory of the user GUEST at the NRM. Users familiar with the Intel development environment, will notice that the OpenNET network translated the output of the DIR command at the NRM to DOS format. The user can change directories at this drive, invoke DOS applications from this drive, if they are stored at the NRM, store data files on this drive. The underlying OpenNET protocol is transparent so all existing DOS applications programs can access this drive just as if it was stored locally.

Any time a user wants to fiind out which of his virtual drives are connected to which servers he enters the NET USE command.

```
C:> NET USE
Local Network
Status Device Name
---------------------------
E:    \\APPS_NRM1\GUEST
F:    \\APPS_NRM2\GUEST
G:    \\APPS_NRM1\GUEST
Command completed successfully.
```

More information on the NET USE commands are given in the OpenNET PC Link manuals.

DISTRIBUTED JOB CONTROL SYSTEM FOR THE PC

The Distributed Job Control system on the NDS-II allows currently idle networked development systems to be supplied to the network as public resources. This is a remote job execution unit to which jobs can be sent by other users on the network. Remote job execution offers higher throughput and increased efficiency, as more than one computer can be controlled by a single user. For more information on DJC, refer to Application Note 244, "DJC A key to increased network productivity". It is recommended that this Application Note be read since a number of concepts explained in the following paragraphs assumes that the user has knowledge of the DJC system at the NRM.

The Network Resource Manager (NRM) is the nerve center of the DJC system. All jobs are scheduled and queued by the NRM. Traditionally the PC user had to wait for his compiles to be finished locally before doing anything else on the PC. With the introduction of the OpenNET network, a mechanism has been designed to let the resources of DJC at the NRM be made available to the PC user. This feature enables the PC user to edit files at he PC and send the compiles over to the NRM. An efficient tracking system has also been designed to help keep the user informed at all times on the status of his job. With the introduction of the 286/310 iNDX based Compile Engine shown in Figure 10, the

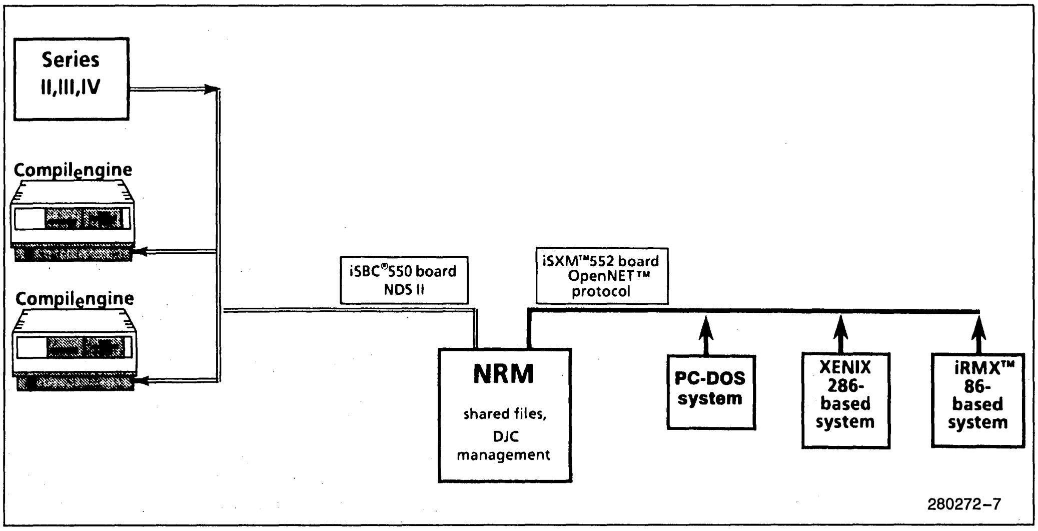


Figure 10. PC's, Compilengine, DJC and the OpenNET™ Systems

```
Reexporter          ;invoke the REEXPORTER utility
Export Reexport to iNDXUTILITY.Q nolog ;now export this job again
```

Figure 11. Contents of REEXPORT.CSD

throughput on such exported jobs increases dramatically. This directly translates into increased productivity and efficiency for the PC user.

The PC-Export package consists of a utility that runs at an iNDX station on the DFS side of the NRM (i.e., NRM itself, Series-IV or 286/310 compile Engine). This package called REEXPORTER.86 checks through a specified directory of all users on the network and if any jobs from the PC exist it will export it to the appropriate queue at the NRM and then will delete the file. Each OpenNET PC user who needs access to the DJC manager at the NRM must create a directory NRMDJC.DIR under his/her HOME directory. It is advisable to limit the above operation to only those users who need to access the DJC system at the NRM. This will help the REEXPORTER utility in having a faster turnaround rate, by not checking redundant directories.

The Superuser then has to create an export file REEXPORT.CSD shown in Figure 11.

It is assumed that the NRM has three queues, 8bit.q (for 8 bit jobs), 16bit.q (for 16 bit jobs) and iNDXUTILITY.Q (for utilities other than compiles, limited to doing system administrative jobs). For more information please read Application Note #244 "DJC a Key to increased network productivity". The Superuser must bring up a Series-IV workstation, or a 286/310 Compile Engine in Import mode, importing from all the three queues. This is shown in Figure 12.

```
>Import from 8bit.q, 16bit.q, indxutility.q
```

Figure 12

The command now puts the workstation as a network resource that all user can acess.

Now export REEXPORT.CSD to iNDXUTILITY.Q

```
>Export REEXPORT.CSD to iNDXUTILITY.Q Nolog
```

Exporting from the PC

To export jobs from the PC, the user must first connect to the NRM using the NET USE command explained in previous chapters. One of the design considerations was to make this utility as easy to use as possible. The following paragraphs illustrate how this has been brought about.

Consider this example file that does a compile and link, COMPILE.CSD. A requisite is that the file must have a .CSD extension, as it is this extension that informs the NRM that it is a command file.

```
;queue = "16bit.q"
lname define 1 for/wini0/libs.dir
lname define p for /wini0/strng.dir
plm86 example.p86 debug optimize (2)
if % status = 0
 link86 example.obj,1/compac.lib,&
   p/hstrng.lib,1/osxcom.lib &
   to example.86 bind
 end
```

The first line in the command file is a comment, which also indicates the queue where this job is to be executed. As far as the DJC manager is concerned, this is just a comment. But the REEXPORTER utility uses this field to find out the queue name. This comment field must exist within the first 128 bytes of the command file. An absence of this field will result in the job not being sent to any queue. The queue name must be enclosed within double quotes.

Since there is no way by which the NRM can access files stored on the PC, all source, libraries and objects must reside on the network. Note the two commands after the comment which set up the logical names for directories used in the job. Doing this will ensure that the right libraries are used.

Now all the PC user has to do is to copy this file to the directory NRMDJC.DIR under their home directory at the NRM. The REEXPORTER utility does the rest. For example if virtual drive G: has been connected to the NRM.

```
C: \Copy COMPILE.CSD G: \NRMDJC.DIR
```

Once the job has been reexported it will be deleted from the directory. This helps the user in determining if the job got exported or not. The REEXPORTER utility also creates a log file in the NRMDIC.DIR directory for each job exported. This allows the user to find out if his job was successful or not. The COMPILE.CSD file will be replaced by a COMPILE.LOG file once the job has been completed. The COMPILE.LOG file is a LOG file of all the operations by the job.

The REEXPORTER utility was designed and implemented to allow the OpenNET PC user access to the powerful Distributed Job Control mechanism at the NRM. The utility has the capability to determine all the users on the network, work out their home directories and look for jobs sent from PC's. On finding a job, the utility determines the appropriate queue and reexports the job to that queue. The status of each job is displayed on the screen at all times. Status information includes username, jobname, queue and the status of the exported job. The use of this utility is restricted to the Superuser. A normal user invoking REEXPORTER.86 will generate an insufficient access rights exception.

REEXPORT.CSD is a batch file configured as a job that runs forever. It first uses REEXPORTER to check for jobs in the directory NRMDJC.DIR of all users and exports them to appropriate queues. It then reexports itself to the same queue. Due to the way the DJC mechanism is structured, the import station will start executing all the jobs found by the REEXPORTER utility, and on completing all of them will execute the REEXPORT.CSD job once again to look for more work to do. The cycle keeps repeating forever.

Referring back to the IMPORT command in Figure 12, highest priority is given to 8bit.q and lowest to iNDX-UTILITY.Q. This way the system manager makes sure that all jobs waiting in the first two queues are executed before the REEXPORT.CSD job is started again. This helps the NRM in controlling the queues, and not overloading them at one single time. A point to note at this time is that the REEXPORTER utility is capable of exporting up to 23 jobs a minute.

The REEXPORT utility combined with OpenNET networking opens out a completely new environment for the PC AT user. An environment where compiles, links and locates can be done remotely. The PC user has at his/her disposal the power of the iNDX Distributed Job Control subsystem. This help in bringing about an increase in productivity that normally could not have been achieved without Intel's OpenNET network. The PC AT user can spend more time on interactive work such as program generation or debugging, while the compiles are being done elsewhere. This feature set should be used by developers doing system designs in todays world where "Time to Market" is key.

Summary

This application note has discussed in detail all the different methods by which the PC AT can be integrated into the Intel Development Environment, from serial interfaces to networking. These different methods can be intermixed to suit your needs. Serial interfaces and disk transfers support the low end needs while OpenNET brings about a powerful new environment to the PC AT. This coupled with the REEXPORTER utility, can help increase the productivity of the PC AT user dramatically.

APPENDIX A THE KERMIT PROTOCOL

THE KERMIT PROTOCOL

The KERMIT protocol is designed around character oriented transmission over serial lines, and incorporates features from decnet, arpanet, dialnet etc.

File transfer takes place over transactions. A transaction is an exchange of packets. A successful transaction is done when one system sends a packet and the remote system acknowledges it.

Transmission begins with a send_init packet(s) and ends with a break_transmission (b) or error (e) packet. All communication is done through packets, even if no data is being sent.

The Kermit packet is built around the following format.

mark	length	seq	type	data	check

All the fields are ASCII characters.

Mark

This is the synchronization character that marks the beginning of a packet. This is normally a cntrl-a and can be redefined.

Length

The number of ASCII characters within the packet following this field.

Seq

The packet sequence number, ranging from 0 to 63. Sequence numbers wrap around to 0 after each group of 64.

Type

The packet type is represented in a single ASCII character. The different packet types are:

```
D data packet
Y acknowledge
N negative acknowledge
S send initiate
B break transmission
F file header
Z end of file
E error
```

Data

The contents of this packet.

Check

A checksum on the characters between, but not including the mark and check. The check for each packet is computed by each host and must be equal for the packet to be accepted.

KERMIT File Transfer Sequence

File transfer is initiated by the sender sending a send_initiate packet, where parameters like packet length, time out limits are specified.

The receiver than sends an ack (y) with its own parameters in the data field.

The sender then transmits a file—header packet which contains the filename in the data field. The receiver then sends an ack.

The sender then sends the contents of the file in data packets (d), any data that is not in the printable range is prefixed and replaced by a printable equivalent. Each d packet has to be acknowledged before the next one is sent.

After all the file data has been sent the sender then sends an eof packet. The receiver acks it.

End_of_transmission packet (b). The receiver acks it and the transaction is over.

KERMIT-MS Commands

CONNECT

The CONNECT command connects the PC as a terminal to the remote system. KERMIT-MS uses either communications port 1 or 2 and uses full duplex and no parity. These can be changed using the SET command. To get back to KERMIT from terminal emulation type in the escape character followed by the letter C. The escape character by default is CENTRL-]. This can be modified by the SET ESCAPE commnd. SET BAUD changes the baud rate, SET PORT changes the serial port. For example:

```
C:\KERMIT <cr>
KERMIT-MS>SET PROT 1    ;select port 1
KERMIT-MS>SET BAUD 9600
KERMIT-MS>C             ;connect
```

SEND

The SEND command causes a file or a group of files to be sent from the local PC to the KERMIT on the remote system. The remote KERMIT must be running in either server or interactive mode; in the latter case the user must have already given a RECEIVE command and escaped back to the PC.

SEND filespec1 [filespec2]

If filespec1 contains a wildcard then all matching files will be sent in the same order that the DOS would show them on a directory listing. If filespec1 contains a single file, the user may direct KERMIT-MS to send that file with a different name.

SEND KERMIT.ASM TEST.ASM would send the file KERMIT.ASM as TEST.ASM

or

SEND *.ASM will send all files with the extension .ASM to the remote system.

Once the SEND command has been invoked the name of each file will be displayed, packets transferred, retries and other counts will be displayed along with other informational messages. If file transfer is successful a "COMPLETE" message will be displayed else an error message will be displayed. When the specified operation has been done the program will sound a beep.

Several single character commands can be given while a file transfer is in progress.

CNTRL-X	Stop sending the current file and go on the next one.
CNTRL-Z	Abort file transfer.
CNTRL-C	Return to KERMIT-MS.
CNTRL-E	Send an ERROR packet to the remote server in an attempt to bring it back to server or interactive mode.

RECEIVE

The RECEIVE command tells KERMIT-MS to receive a file or a group of files from the remote KERMIT. KERMIT-MS simply waits for the file or files to arrive. The user should have already issued a SEND command at the remote KERMIT and escaped back to the PC before issuing the RECEIVE command.

Syntax:

RECEIVE filespec

If the optional filespec is provided the incoming file is stored under that name. The filespec may include a device designator or may consist only of a device designator. For example:

RECEIVE TEST.ONE	;will name the incoming file as TEST.ONE
RECEIVE A:TEST.ONE	;will name the incoming file as TEST.ONE and store it in drive A:
RECEIVE A:	;will store all incoming files in Drive A

If an incoming file does not arrive in its entirety, KERMIT-MS will normally discard it. This may be changed by the SET INCOMPLETE KEEP command, which will keep as much of the file that arrived successfully.

If the incoming file has the same name as a file that already exists and "WARNING" is set ON, KERMIT-MS will change the incoming file name and inform the user of the new name. If WARNING has been SET OFF using the SET WARNING OFF command, files with the same name as incoming files will not survive.

SET

The SET command allows the user to modify various parameters for file transfer and terminal emulation. These parameters can be displayed with the SHOW command.

APPENDIX B
SETTING UP OpenNET™ CONNECTIONS BETWEEN PC & NRM

```
APPS_NRM1      :address = 0x80000a0100000000aa00003510000000
APPS_NRM2      :address = 0x80000a0100000000aa000034de000000
APPS_XENIX_1   :address = 0x80000a0100000000aa00002de2000000
DIAG_NRM       :address = 0x80000a0100000000aa00000c0f000000
MFNG_RMX_1     :address = 0x80000a0100000000aa000006e4000000
MKTG_NRM       :address = 0x80000a0100000000aa00000304000000
APPS_PCAT_SS   :address = 0x00010a0100000000dd00002584000000
```

Figure 13

OpenNET™ CONCEPTS

To enable the PC to talk DFS-OpenNET protocol, the user must install a PC-Link card in the IBM PC, and the networking software PC-LINK supplied by Intel. The prerequisite is that the PC should have PC-DOS Version 3.1 or greater. Follow the installation instructions given in the PC-Link manual. It is advisable to create a directory on the PC called COM (short for Communication S/W) and install all the PC-LINK software in this directory.

Once the hardware and software have been installed on the system the user can now use the PC as a consumer off of the NRM. Before going into a discussion on the actual use, a number of concepts have to be explained, to give the reader a better understanding of how PC's work when networked to the NRM using the OpenNET protocols.

The NETADDR File

Each computer on the network is assigned a name, to identify it. These computers can be named anything, but it is preferable to have one standard naming convention. This makes it easier for the users to connect up to the server of choice. The NETADDR file is a text file that contains the names of these servers and their addresses on the network. This is extensively used by the PC-LINK software to connect to the desired server. This gives the user the flexibility to connect to any server just by giving its name, and the PC-LINK software automatically directs the connection to that server. A sample NETADDR file is shown in Figure 13.

The NETADDR file above has the lists of 6 servers on the OpenNET network. Note the naming convention. The first four letters in the name indicate the department where the server is stationed. For example APPs is short for Applications Engineering, DIAG for Diagnostics Engineering, MFNG Manufacturing etc. The rest of the name indicates the type of server (NRM, XENIX or RMX). This naming convention helps in connecting up to the different servers without any confusion. This file is created by each individual PC consumer using any standard text editor (one which does not put control characters in the file). One of the entries in the NETADDR file is the name of the users PC where this file resides. This is important as the PC-LINK software cannot be invoked without this information. The name and ethernet address of the consumer station is necessary for the NRM OpenNET file server to send message packets back to the correct originating consumer (i.e., the NRM should know which consumer station has requested for a particular process for sending back the response to it).

The rest of the numbers following each computer name are the Port address and the iSXM 552 Ethernet address of the server. An example is shown in Figure 14.

The Ethernet address of the iSXM 552 is obtained from the NRM on boot up. Refer to the Chapter "Installing the iSXM 552 in the NRM" for further details.

The MSNET.INI File

The MSNET.INI file is the PC-LINK initialization file for setting up the help files and loading the PC-Link

APPS	is the department where the server is located
NRM2	is the NRM name (2 is used to differentiate it from the other NRM)
80	is the port address (always 80)
00aa000034de	is the ethernet address of the iSXM552 board at NRM2.

Figure 14

board with the communication software. Any time a PC station is brought up as a consumer this file is parsed and executed. The file may need a little modification if the communication software is located in some other directory. The sample MSNET.INI file is used as an example, see Figure 15.

Virtual Drives

PC-LINK uses the concept of virtual drives to connect to the NRM file server through the OpenNET network. When a connection to a NRM server is made the PC-LINK software creates a virtual drive, which is identical to the PC drives. The only difference is that it does not physically exist on the PC. The user treats this as any other drive on the PC. The number of virtual drives that can be used is limited to the English alphabet (26). The user can connect different virtual drives to different directories at the NRM file server and all these drives can be used just as if they existed on the PC.

Configuring PC Link

PC Link as shipped uses default settings for its memory window, number of connections, interrupt levels and number of servers that it can access. The following paragraphs will discuss various configuration parameters that will allow the user to configure PC link to suit his environment.

CONFIGURING THE BASE FOR THE PC LINK BOARD

All communication with the PC link board takes place via a dual-port memory, which is a 32K window that may start on any 64K window within the first megabyte of addressable memory. The default base is 0A000h. In some cases the window might clash with an existing board or application. This default can be changed by jumpers on the PC link board and by making modifications to the MSNET.INI file. Figure 16 indicates all possible bases and associated jumpers.

After the jumpers have been selected, the MSNET.INI file has to be changed to inform the PC link software about the new window being used. Referring to Figure 15:

```
start redirector $1
start rdr $1
 \command.com/c type \com\pclink.msg
 chknet
 xport/sys:at/base:d
   /file:c:\com\ubcode.mem
 session \com\netaddr
 redir
 setname $1
 \command.com/c psclose
```

The string of commands following the start redirector $1 or start rdr $1 indicate the sequence of events in

```
use $*
   use $* /*

print $*
   printq $*

name
   setname

start redirector $1
start rdr $1
   \command.com /c type \com\pclink.msg
   chknet
   xport/sys:at /base:d
    /file:c:\com\ubcode.mem\nvcs:5
   session \com\netaddr
   redir
   setname $1
   \command.com /c psclose
```

Figure 15. (Sample MSNET.INI file)

loading the PC link board and starting the network. XPORT is the network program that loads the PC link board and the driver. One of the options that can be specified in the xport commands is the base option. In the previous example the base was set to D. This base should reflect the base at which the PC link board is strapped.

Configuring Connection Limits

PC link allows the user to have simultaneous connections to a number of file servers which is configurable.

The PC link defaults allow the user to connect to two file servers simultaneously and have up to 5 active virtual circuits. The MSNET.INI file supplied with PC link has to be modified for users who need access to more than two file servers and more simultaneous connections.

For example:

A user needs connections to three different servers. The first two NET USE commands will come back successfully, however the third command will come back immediately with a message "Connection Refused". This is due to the fact that the PC link software uses a default value of 2 for the number of servers it can simultaneously access.

Starting Address	E5	E8	E11	E14
00000	E4	E7	E10	E13
10000	E4	E7	E10	E15
20000	E4	E7	E10	E13
30000	E4	E7	E10	E15
40000	E4	E9	E10	E13
50000	E4	E9	E10	E15
60000	E4	E9	E12	E13
70000	E4	E9	E12	E15
80000	E6	E7	E10	E13
90000	E6	E7	E10	E15
A0000 (DEFAULT)	E6	E7	E12	E13
B0000	E6	E7	E12	E15
C0000	E6	E9	E10	E13
D0000	E6	E9	E10	E15
E0000	E6	E9	E12	E13
F0000	E6	E9	E12	E15

Figure 16

```
C:\ NET USE H: \\APPS_NRM\JOHN PASSME
Command completed successfully
C:\ NET USE I: \\DEMO_NRM\JOHN PASSME
Command completed successfully
C:\ NET USE LPT1: \\DIAG_NRM\JOHN PASSME
Command Refused
```

In the above case the user tried to access more than one server and since the default was set at 2, the PC link network management facility came back with an error. To change this default value, edit the MSNET.INI file and modify the REDIR specification to read:

```
REDIR /S:5   ;connection available for up to
5 servers
```

Reboot the PC and PC link will now allow the user to connect to up to 5 servers simultaneously.

Another variable that can be configured is the number of active virtual circuits at the PC. As a default PC link will allow up to 5 net uses (Logons) to a server. Again when the default limit is exceeded the system will come back with a "CONNECTION REFUSED" message. For example consider the following NET USES:

```
C:\ NET USE F: \\APPS_NRM\SRIVATS PASSME
Command completed successfully
C:\ NET USE G: \\APPS_NRM\AP1 PASSAP1
Command completed successfully
C:\ NET USE H: \\APPS_NRM\SY1 PASSSY1
Command completed successfully
C:\ NET USE I: \\APPS_NRM\SYO PASSSYO
Command completed successfully
C:\ NET USE J: \\APPS_NRM\APO PASSAPO
Command completed successfully
C:\ NET USE K: \\APPS_NRM\JOHN PASSME
Connection Refused.
```

In the above example the number of simultaneous connections was set at a default of 5. Modify the REDIR command to include the following option.

```
REDIR /S:5/L:10   ;connection available for up to 5
                   servers and 10 simultaneous con-
                   nections.
```

Reboot the system and PC link will allow the user to connect up to 5 servers with up to 10 simultaneous connections.

APPENDIX C INSITE LIBRARY PROGRAM

MS-KERMIT and ISIS KERMIT are available from:

Intel Corporation
2402 West Beardsley Road
Phoenix, Arizona 85027

ATTN: Insite User's Program Library

Telephone: (602) 869-3805

This is a public domain software and is available for a nominal charge of $25 each.

APPENDIX D NDS-II/SERIES IV TOOLBOX V2.0

NDS-II/SERIES-IV TOOLBOX V2.0

The NDS-II/Series-IV Toolbox V2.0 is a set of 7 diskettes with useful programs for NDS-II/Series-IV and OpenNET users. The programs used from this toolbox were MSCOPY.86, ReExporter, NET CONNECT. It contains many other useful utilities. An index listing the various programs in the Toolbox are listed below.

NDS-II/Series IV Toolbox 2.0

Chapter 1—CONNECT 1-1
Chapter 2—NDS-II to NDS-II Communications. 2-1
Chapter 3—TREE 3-1
Chapter 4—MENU COMPILER 4-1
Chapter 5—MSCOPY 5-1
Chapter 6—NETWORK CP/M-80............ 6-1
Chapter 7—BOOTUP 7-1
Chapter 8—SERVER 8-1
Chapter 9—PRINCE 9-1
Chapter 10—PRMSLO 10-1
Chapter 11—SLEEP 11-1
Chapter 12—ID 12-1
Chapter 13—MDS-800 FPORT................13-1
Chapter 14—DBLIST.........................14-1
Chapter 15—REMOTE 15-1
Chapter 16—PC REMOTE 16-1
Chapter 17—REPORT.......................17-1
Chapter 18—DIRT 18-1
Chapter 19—VIEWPASS.....................19-1
Chapter 20—FDUMP 20-1
Chapter 21—CLOCK 21-1
Chapter 22—IFILES.........................22-1
Chapter 23—LIST...........................23-1
Chapter 24—TA 24-1
Chapter 25—MAILMAN 25-1
Chapter 26—CHECKEXIST..................26-1
Chapter 27—CHECKTIME 27-1
Chapter 28—BVCLIB 28-1
Chapter 29—UDXCOM.LIB..................29-1
Chapter 30—OSXCOM.LIB 30-1
Chapter 31—BVOSX.LIB 31-1
Chapter 32—XID 32-1
Chapter 33—REEXPORTER 33-1
Chapter 34—XTAR 34-1
Chapter 35—ISIS ENVIRONMENT 35-1
Chapter 36—OAP 36-1

intel® **ARTICLE REPRINT** **AR-204**

Technical articles

Smart link comes to the rescue of software-development managers

Resource-management hardware and software join existing development systems into an Ethernet-based network that eases software creation and control

by James P. Schwabe, *Intel Corp., Santa Clara, Calif.*

□ A strong lifeline in a sea of complexity, the new NDS II network development system will help manage the writing of complex software for tomorrow's powerful microsystems. It builds on existing Intellec development systems and the specifications of the Ethernet protocol to create a local network for distributed software development.

Considerable intelligence is contained within the NDS II system, linking programmers' work stations and managing the interactive flow of software development that results. Communications control, via Ethernet or an even simpler alternative, is split between the central manager and the work stations.

At the heart of the system is the network resource manager, which both controls the net of work stations and lets the user configure it to suit the development task under way. The NRM will also manage a powerful system memory of Winchester-technology disk drives.

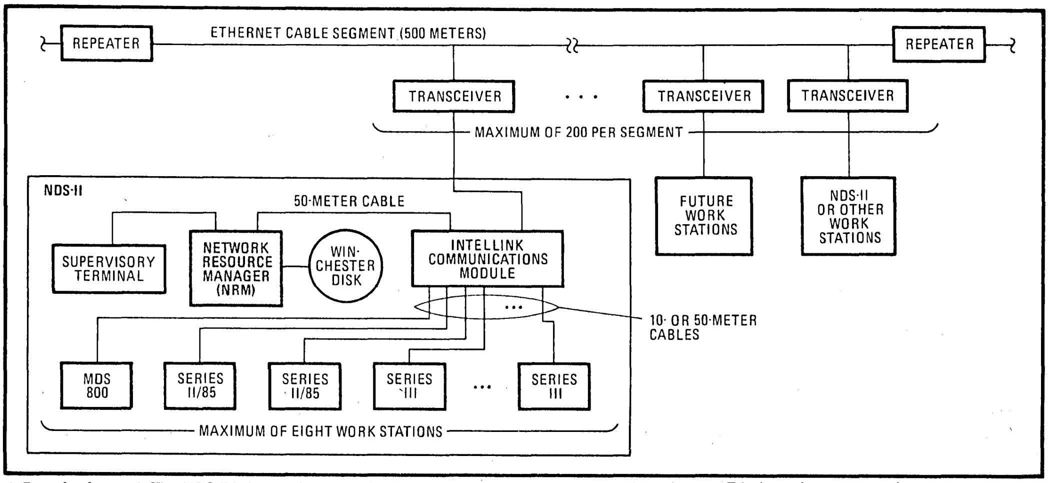
The manager itself is an example of the boons of well-thought-out and complex software, for it contains powerful system tools. Among these features are a hierarchical file structure that is also distributed and a file-protection setup that offers the maximum flexibility in access to files while guaranteeing their integrity.

Important program-management tools include a routine that oversees the rewriting of software during development and another that automates the generation of a complete program from the most current modules.

The NDS II is the second step in the evolution of Intel's network architecture, iLNA [*Electronics*, Aug. 25, 1981, p. 120]. It connects Intellec development systems together so they can share large-capacity Winchester disk drives and a line printer located at the NRM. It will also serve as the basis for a whole new line of modular development system tools such as remote emulators, logic analyzers, and more.

Both the NRM and each work station can be connected directly to the Ethernet coaxial cable by a transceiver or by the Intellink communications module (Fig. 1). By itself, the Intellink module provides nine ports for interconnection, creating a local network of nine systems (eight work stations and one NRM). To another controller, the Intellink represents a segment of Ethernet cable that has nine transceivers already in place and working.

For networks with a radius of 50 meters or less, Intellink is a simple, low-cost alternative to installing Ethernet cabling and transceivers. Any work station can



1. Developing net. The NDS II brings existing Intel development systems, or work stations, into an Ethernet. A new network resource manager and the Intellink communications manager make management of distributed software development possible.

be installed by simply plugging a 50-m transceiver cable directly into the Intellink—a 5-second operation.

For expansion beyond nine systems or to a distance greater than a 50-m radius, the Intellink provides a built-in port for connecting the local cluster to Ethernet cable by means of a transceiver. Connection to the Ethernet allows communication with other work stations, NDS II networks, or other Ethernet-compatible devices that use the iLNA network architecture.

No matter which physical setup is chosen, each work station has independent access to, and can be directly accessed from, the Ethernet and the NDS II network. Each has a unique work-station identifier, distinguishing it from every other terminal in the world and ensuring correct communication between stations on the various local networks.

For multiple-net environments, each network can have a unique network identifier to allow their coexistence on one Ethernet. In a single net, the network identifier is not used, but its assignment ensures an orderly progression to a multi-net environment.

All current Intellec development systems can be upgraded to NDS II work stations. An upgrade consists of a communication-controller board set, software, and either 10- or 50-m cables.

The communication controller, a two-board set that plugs into any Intel Multibus chassis, provides many of the data- and physical-link functions of the six-layer standard reference model for open-systems interconnection (Fig. 2). The data-link functions performed are framing, link management, and error detection. Physical-link functions include preamble generation and decoding and bit encoding and decoding.

One board contains a 5-megahertz 8086 microprocessor with local random-access and read-only memory and interval timers, as well as direct-memory-access channels for sending and receiving data at 10 megabits per second. The second board contains bit-serial send-and-receive logic, packet address-recognition logic, and error-detection logic. The boards ensure that bad packets resulting from a collision are ignored.

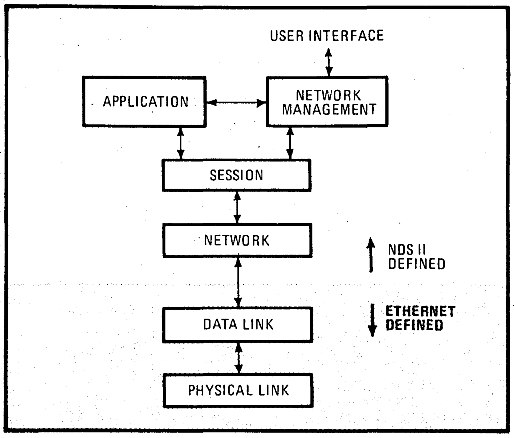
The NRM coordinates all the work stations' activities and manages file access to the shared disks. Initially, it will support one 8-inch 35-megabyte Winchester disk subsystem, as well as Intel cartridge-module disks. Multiple-disk support is in the wings, along with a larger 84-megabyte disk. It will be possible to attach six disks to one NRM, providing more than enough on-line shared storage for large program development and archiving. In addition, each work station can contain 2.5 megabytes of floppy-disk storage as a local resource.

Control contingent

The NRM (Fig. 3) comprises 13 Multibus slots, power supply, 8086-based system-processor board, input/output board based on the 8088 and 8089, 512-K-byte memory board with error checking and correction, two communication boards, and one 5¼-in. floppy-disk drive. The cabinet also has space for a cartridge-tape unit, expected to be delivered in mid-1982, which will give full intelligent archival backup for the Winchester disks housed in the attached cabinet.

To protect the integrity of the network, access to the NRM is restricted: a special supervisory terminal connected to the unit's serial port provides an interface with its commands and utilities. These facilities include system generation, intelligent archiving, and normal network maintenance such as the creation of any necessary user identifications.

The most important utility for system configuration is called Sysgen, an interactive routine designed to assist the supervisor, or project manager, in creating the NRM operating system. Sysgen makes it possible to create, modify, or delete system parameters, peripheral-devices configuration, and network configuration. It allows the project manager to tailor the network configuration on the fly in order to fit the changing needs of microprocessor development projects.



2. New layers. To the hardware layers of Ethernet, NDS II adds software layers that permit up to eight users to work together. The network layer need not be present if NDS II is not linked to the Ethernet, simplifying the operating system.

From the work-station perspective, the NRM is a remote file system. Each station functions as a standalone development system for all tasks not requiring NRM resources. When access to these resources is required, the user simply logs onto the network. The work station's resident operating system formats the appropriate file request, which the NRM processes interactively with other stations' demands.

The NRM operating system is multitasking, allowing a work station to access a file on the shared disk while other stations concurrently access other disk files. The interleaving of disk accesses, as well as the high-speed packet transmissions on the Ethernet, enables each work station to share equally the large file store—its being accessed by one user does not prevent other work stations from gaining access.

In an eight-station environment, the performance degradation due to network contention and the NRM operating system will be no more than 10%. This performance is one of the major reasons why distributed development systems provide a more cost-effective method for microprocessor development than time-shared systems; the former are much less susceptible to saturation under concurrent loading than are the latter.

Managing the work

To ensure efficient software development, high performance must be combined with tools to manage software complexity. For example, large software projects are often broken down into small tasks, and efficient file sharing becomes essential to project coordination. The shared-file system on NDS II is built on the RMX-86 volume-based hierarchy in which each user directory represents a node on a hierarchy of directories, commonly referred to as a hierarchial file system (Fig. 4).

Hierarchical file systems can contain a multitude of directories and data files. At the apex is the root volume, a conceptual file from which all directories emanate. The root volume contains all the volumes of the directories. Each volume can contain as many directories or files as available disk space will allow, and any directory may contain other directory files or data files. Each file (directory or data) can be traced through the hierarchy by its own path name. The NDS II hierarchical file system goes one step further by extending from the NRM to include the directories at the user's work station. When the user logs off the network, the only directories available are those on the work-station disks. When the user logs on, he or she gains access to the NRM system directories.

Thus each programmer has access to a common data base without the confusion of sifting through one massive directory. What's more, the structure keeps other users' files out of the way. In addition, it permits logically separate types of software within a user's directory. A programmer can create subdirectories to separate source files from object files, from backup files, and so on.

As a project's size increases, the number of directories and the complexity of path names in the system also increases. To simplify the task of accessing any particular directory, the user can assign a less cumbersome name—what amounts to a macroinstruction. Then, the user simply types in this macroname. Maximum flexibility is maintained, as each programmer can assign macronames to any directory.

An added benefit from macroname assignment is device transparency: the user concerns himself only with directories, irrespective of physical location. Physical devices are fixed in size and location, as opposed to directories, which can be adjusted to organize the contents in an optimal fashion.

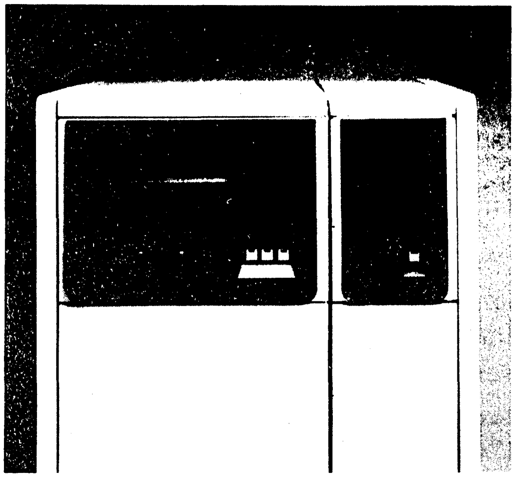
File protection

Before accessing the network, each user must be identified to the NRM through a log-on procedure. This setup establishes a unique user identification that is subsequently used to control access to files and directories in the hierarchical file system. Each directory and data file has specific "owner" and "world" access rights, which protect against accidental modification or deletion.

A file has three possible access rights for both the owner and the world: read, write, and delete. A directory also has three similar access rights for both the owner and the world: list a directory, add a directory entry, and delete a directory entry.

The access rights in file systems improve coordination during software development by allowing complete modules that have been tested and debugged in a user's work space to be converted into read status for the world. Then these modules can be integrated and tested with other independently developed software modules. Thus modules declared as read-only are guaranteed to be the most current debugged versions, and a common data base of completed modules is ensured.

Extended to multiple-project environments, the file system can provide logically separate work spaces for each project group. Specific directories can be set aside for complete modules for various projects. Each user can develop portions of the program in a private work space with guaranteed file protection and can use the public files (or directories) for integration and testing of the



3. Manager. The network resource manager (NRM) in the cabinet's left side governs access to the 35-megabyte Winchester drive on the right. Access to network-managing software is gained only through a supervisory terminal attached directly to the NRM.

module under development. Commonly used utilities and compilers can be accessible in a specific directory as public files (read-only for world access) to eliminate the necessity of redundant files at each work station. As a result, all programmers can proceed without fear of inadvertent modification of private files either by others or by themselves.

As well as managing communications between shared disks and work stations, the NRM maximizes the use of all network resources with distributed job control. DJC allows the user of any work station to export a batch job to the NRM for remote execution.

To accomplish this, the NRM classifies each work station into one of two groups—private and public. It keeps track of the public work stations and uses them to execute the queue of batch-type jobs. A user can declare any work station as public: available for use by the NRM for remote execution. Also, a programmer can send a job to a specific queue at the NRM by using the export command. The NRM executes the job on a public work station and return the results to the user directory.

With DJC, the resources of the entire network can be shared to maximum advantage. A typical project involves program-module editing and debugging at Intellec series II or model 800 work stations, while a 8086-based Intellec series III unit can provide a host execution environment to compile completed modules quickly. DJC allows the user to export the compilation process to the high-performance series III work station, then return immediately to other tasks while the NRM oversees the compilation. At any time, the users can check on job status or queue status by typing a command from their work stations.

New work stations

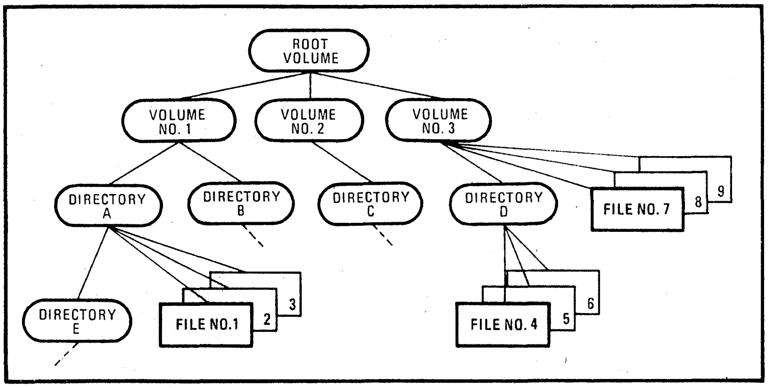
Currently, Intellec development systems provide a single-task environment and therefore can be declared public to the NRM as users finish on-line work. Later this year, Intel will introduce high-performance work stations with foreground-background capability to allow a user to run a job in the foreground while making the background public so that jobs exported by other programmers can be executed through DJC. Foreground-background capability with DJC will effectively double the usefulness of the work station and substantially cut the cost of development time.

In-house benchmark tests indicate that the performance of each work station connected to the NRM is much improved. For example, a compilation executed with all file requests from the NRM hard disk is twice as fast as requesting files from the work station's floppy disk. Each station enjoys hard-disk performance during compilation, assembly, and any file manipulation—at a fraction of the cost of a dedicated disk system.

User's tools also speed program development, as well as make management easier. The most important programmer tools on NDS II are SVCS (software-version control system) and MAKE, an automatic software-generation tool. They provide a superset of the functions offered by the SVCS and MAKE found in the Unix programmers workbench.

SVCS controls and documents changes to software products, handling both source and object files. It contains facilities for storing and retrieving different versions of a given program module, for controlling update privileges, and for recording who made what changes, when, and why.

Documentation of module status and of the levels, or versions, involved is the key factor determining the success of program development by group effort. Valu-



4. Climbing an inverted tree. To find a file in the NDS II, the user first goes to the root volume of this hierarchical file structure. From that volume, he or she can go to the project volume assigned by the project manager and access other directories or files that have been declared accessible.

MAKEing it easy to revise programs

NDS-II's MAKE facility is a development tool for both generation and documentation of a software system. Suppose, for example, a software system called PGM.86 consists of three separate programs linked together, and, for simplicity, that each program consists of only one compiled source file, rather than a subsystem of multiple files. This relationship forms a dependency that would be graphed by the user as in the figure below.

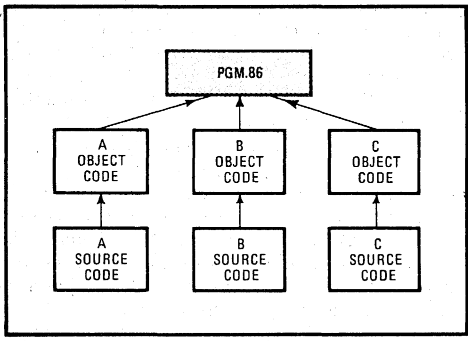
With the MAKE facility, a user can create an automated-generation procedure for the system PGM.86 that checks the currency of each subprogram. A MAKE command file that does so is illustrated in the accompanying table.

When the command file is invoked, the commands it contains are executed in top-down fashion. In step 1 of the table, the facility first checks if the PGM.86 is older (represented by the greater-than sign) than any of its dependent object-code modules. The facility checks and compares the date and time stamp of each module with that of PGM.86. Date and time stamps are updated automatically whenever a file is modified.

If any of the object modules are newer versions, then MAKE is instructed to link together the latest versions of the object modules to form the latest version of the software system. Before executing the link routine, the MAKE facility must first check to see if any of the object files are older than the related source files given in the dependency graph, as shown in steps 2, 3, and 4.

The MAKE facility goes through each step and executes the specified task only if the specified condition is true. Once the dependency graph is created, the MAKE facility can quickly and automatically generate the latest version of a software system under development even when source files change frequently.

The MAKE facility removes much of the guesswork surrounding software-system generation by ensuring the latest versions of source code is incorporated into the final software system. The dependency graph in its current form can also be printed by NDS II to document the software-system construction without having to keep an out-of-date sketch taped to the laboratory wall.



MAKE PROGRAM FOR PGM.86

Steps	Statements

1	IF PGM.86 > A.OBJ, B.OBJ, C.OBJ THEN
	RUN LINK86 A.OBJ, B.OBJ, C.OBJ TO PGM.86
	END

2	IF A.OBJ > A.SRC THEN
	RUN PLM86 A.SRC
	END

3	IF B.OBJ > B.SRC THEN
	RUN PLM86 B.SRC
	END

4	IF C.OBJ > C.SRC THEN
	RUN ASM86 C.SRC
	END

able development time can be lost trying to work someone else's modified modules if documentation specifying what, where, when, and why changes were made is not available. In fact, as programs become more complicated, even the module writer may not exactly remember the history of the module.

Automatic documentation

SVCS provides a tool for automatic documentation of these facts. When a new module is created, it is set to level 1. All subsequent versions of the module are maintained with in a single file. Changes to the module are stored as "deltas" to the original. SVCS automatically records what changes were made and when they were made, and it requires the modifier to specify a reason for the change. The project manager may create a software checkpoint at any time by declaring the module as the next release level; subsequent deltas will then be applied to only this new release level.

Other capabilities in SVCS also increase project control. Restrictions may be placed on who is allowed to make changes to which modules and at which levels. An identification facility is also included, allowing the system to stamp modules containing object code with version information. From this information alone, a user can determine the level of source code used to generate the object module and thereby determine exactly which level of software is current and which level is being executed. To aid support groups in future maintenance of the program, any level of a software system can be regenerated from the original modules.

The second important program management tool on NDS-II is called MAKE, (see "MAKEing it easy to revise programs," above). When MAKE is invoked, a software system is automatically generated from the most current version of specific modules delineated by a dependency graph. MAKE ensures that the software generation is current and correct, while recompiling only program modules that need to be updated. To coincide with the concept of modular program development, any component of a MAKE could invoke another MAKE to generate a lower-level component such as a library. □

IEEE SPECTRUM

March 1986

Helping Computers Communicate

JOHN VOELCKER

Helping computers communicate

The Open Systems Interconnection model promises compatibility for a variety of computer systems, although not all its functions are yet defined

Computers made by different companies ordinarily do not "talk" to one another. This aloofness sometimes applies even to different types of computers made by the same company. And when it comes to computerized systems, like machine tools and automatic teller machines, the communications problems can be nightmarish. Aside from expensive customized adapters and software links, compatibility remains an elusive target of the computer industry.

Even when two computers or computerized systems can be made to talk to each other, problems may arise in getting networks—whether telephone, satellite, or microwave—to handle the conversation. Will universal compatibility among computers, computerized equipment, and communications networks ever become reality?

Many industry leaders believe that the Open Systems Interconnection (OSI) model is the key to making users' dreams come true. The set of OSI standards being developed by the International Organization for Standardization (ISO) in Geneva, Switzerland, is a framework for defining the communications process between systems. It includes a Reference Model, with seven layers that define the functions involved in communicating; and definitions of the services required to perform these functions.

To implement the OSI model, the ISO also describes protocols —specifications for how information is coded and passed between parties in a communication. Only protocols can actually be implemented; both the Reference Model and the service definitions are merely structures for discussing the functions involved in communications between dissimilar equipment.

Standards are emerging gradually

Computers, computerized equipment, and communications networks are all covered by OSI. This has created considerable confusion among users, because systems that manufacturers claim "conform to OSI" may not necessarily be compatible with other systems for which the claim is made. What the label means is that the equipment uses some of the OSI standards—a subset of those that have been defined so far. Testing for OSI conformance is just beginning.

Many of the protocols to implement OSI are now complete, and some manufacturers offer products to implement various of these standards. The ISO will continue to expand the functions covered by OSI as new communications network architectures and technologies emerge. But the revisions are being made, the architects say, so that older equipment will not be rendered obsolete.

Some companies—General Motors and the Boeing Co., for example—already specify the use of OSI protocols in certain computer networks. IBM is examining how it can make its own computers and network standards communicate with OSI equipment. Digital Equipment Corp. has announced that within three years, it will replace proprietary protocols in its DECnet network with OSI protocols. A number of suppliers have banded together in a new group, the Corporation for Open Systems, to promote acceptance and use of OSI protocols. In short, the outlook for OSI is promising.

No one contends, of course, that all computerized equipment should be covered by OSI. There seems little need, for example, to allow the microprocessor in a new refrigerator to communicate with the international banking industry's funds-transfer network. But in many industries, the ability to interconnect many different computer systems and communications networks could radically improve the way business is done.

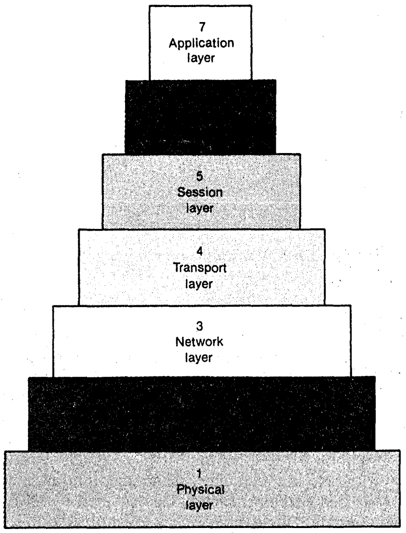
Frustrations abound

Consider the plight of a design engineer who must use a newly installed computer-aided engineering (CAE) system to analyze the deformation of a cylindrical strut with a load applied. The strut was designed on an older computer-aided design (CAD) system made by another company; the system uses a different graphic descriptor language to represent cylinders than the CAE system does. The design engineer must enter the description of

John Voelcker Associate Editor

Defining terms

Application process: a part within an open system that processes information and uses OSI communication services to communicate with other application processes in other open systems.
Channel: the part of a communications system that connects a message source to a message link; a path for electrical transmission between two or more points.
Conversation: an interactive exchange of information between two systems or systems users.
Function: an action performed to further the communications process by parts of a communications system.
Gateway: devices or systems to connect different network architectures having different protocols by providing protocol translation. Gateways may use all seven layers of the OSI model, but must include Layers 1 through 4.
Layer: a set of network-related services within an OSI networking architecture.
Open system: a computer processor or set of connected processors, for which standards are published, that allows an application running in the system to communicate with other applications in the same or other systems.
Packet-switching: the use of software to route messages dynamically from source to destination within a communications network.
Protocol: a specification for coding messages exchanged between two communications processes.
Service: a function offered by some part of an open system to communicating application processes.



[1] The OSI Reference Model breaks the process of communicating into an orderly sequence of seven layers.

the strut into the CAE system to analyze it. But if he makes any specification changes, he will have to enter them into the old CAD system, which will ultimately generate the drawings to produce the part.

These drawings will be sent to the machine shop, where a prototype part will be produced on a digitally controlled lathe. Once again, the part description must be entered into a computer terminal—that of the machine tool—before the next step in the production process can be completed. If all of these machines could communicate, the engineer could change the specification and transmit the design automatically to the machine tool.

The systems designer who must connect automated office equipment from several manufacturers faces a similarly challenging task. While compatible personal computers can be connected to one of the many local-area networks, other types of computers are not so easily attached.

An IBM mainframe used for accounting and corporate record-keeping, for example, cannot easily exchange information with the company's personal computers. Even the physical media for data storage differ—large tape drives for the mainframe, 5¼-inch floppy diskettes for the personal computer.

To "hardwire" the personal computers to the mainframe would require, among other accommodations, the ability to translate every file from one character set to another. The same obstacle applies to most minicomputers, which might be used for other applications like inventory control in a small warehouse.

This lack of compatibility has remained essentially unchanged for at least 20 years. Families of computers or computerized equipment from a single manufacturer can be connected by proprietary communications protocols, but this ties users to a particular supplier's equipment, locking out competing manufacturers.

OSI to the rescue

The OSI model offers a way to establish unity in the fragmented computer and communications fields. It provides a framework for connecting open systems, allowing any supplier to construct a system that communicates with another made by a different company.

Richard desJardins, chairman of the ISO subcommittee responsible for OSI, notes, "The OSI Reference Model simply describes the many functions involved in a communication between two computers or systems, and the terms used to define those functions."

Implementations of these functions consist of software written to span the gap between the application process, which starts the communication—a program in an automated teller machine, say, that responds to a customer's balance request—and the physical medium over which the communication travels—the bank's private telephone lines, in this case. Often this software is embedded in special-purpose circuitry that is included in computers or other communications equipment.

The physical medium is simply the "channel" over which the message is sent. It includes not only the wires in a telephone system, but also transmitting and receiving stations for satellite and microwave communications, as well as local-area networks.

In each layer of the Reference Model, major functions have been defined. International standards define the services and the protocols to implement them. The OSI Reference Model, defined by ISO 7498, is complete and was adopted by the ISO in 1984 [Fig. 1].

The bottom layer of the Reference Model, Layer 1, is called the Physical Layer. It includes the functions to activate, maintain, and deactivate the physical connection. It defines both the functional and procedural characteristics of the interface to the physical circuit; the electrical and mechanical specifications are considered to be part of the medium itself.

Layer 2, the Data Link Layer, covers synchronization and error control for the information transmitted over the physical link, regardless of the content. This can be thought of as "point-to-point error checking."

Layer 3 is the Network Layer. Its functions include routing communications through network resources to the system where the communicating application resides; segmentation and reassembly of data units; and some error correction.

The Network Layer acts as the network controller by deciding where to route data—either out along a physical network path or up to an application process. Data routed between networks or from node to node within a network requires only the functions of Layers 1 to 3 [Fig. 2]. The network node is called a relay system.

End-to-end reliability ensured

The Transport Layer, Layer 4, includes such functions as multiplexing a number of independent message streams over a single connection when desired, and segmenting data into appropriately sized units for efficient handling by the Network Layer. Through these functions, it compensates for differences in the network services that have been provided. It also provides end-to-end control of data reliability, regardless of the type or quality of the network used.

The functions of Layer 5, the Session Layer, are to manage and synchronize conversations between two application processes. Data streams, for example, are marked and resynchronized to ensure that dialogues are not cut off prematurely. The layer provides two main styles of dialogue: two-way alternating (half-duplex), in which two parties alternate in sending messages to each other; and two-way simultaneous (full-duplex), in which two parties may send and receive at the same time.

The Session Layer's control functions are analogous to the use of control language to run a computer system. While Layer 5 selects the type of service, the Network Layer chooses appropriate facilities and the Data Link Layer formats the messages.

Layer 6, the Presentation Layer, ensures that information is delivered in a form that the receiving system can understand and use. The format and language (syntax) of messages can be determined by the communicating parties; the functions of the Presentation Layer translate if required. The meaning (semantics) of the message is preserved. If, for example, one application process transmitted a file in ASCII code, while another used IBM's EBCDIC, the two sides would negotiate which encoding to use and which side would perform translation.

The top of the Reference Model, Layer 7, is the Application Layer. To support distributed applications, its functions manipulate information. It provides resource management for file transfer, virtual file and virtual terminal emulation, distributed processing, and other functions. It is the layer that will contain the most functionality, and it is certainly the one in which the widest variety of work is being done at present.

Viewed as a system, the layers of the Reference Model can be broken into two groups. The bottom three layers—Physical, Data Link, and Network—cover the components of the network used to transmit the message. The top three layers, however, generally reflect the characteristics of the communicating end systems. Their functions take place without regard for the physical medium actually used, whether it is a satellite, an X.25 network, or a local-area network (LAN). Only the two parties to a communication invoke the functions of the Session, Presentation, and Application layers. The Transport Layer acts as the liaison between the end system and the network.

Freedom of services provided

At each layer of the Reference Model, there are services to carry out the functions. For instance, a service such as requesting initialization of a conversation is needed to initiate the control function for a conversation between two end systems.

The services defined for each layer are performed by building on services provided by the layer directly underneath. Conversely, the services at each layer are called upon by those at the next higher layer. Thus when an application process initiates a communication, it passes its message down through each layer. The functions of each layer add value by providing services that enable the communication to be completed.

One shining feature of OSI is that these service definitions are independent. In other words, any service can be implemented regardless of the methods used to implement services in the layers above and below it. Error checking, for instance, may be provided in different systems by dissimilar devices or by unique software. As long as the device or software provides the service defined by OSI, using an approved protocol, it will perform the same function for the end user.

Specifying independent services allowed protocols for several layers of the model to be developed in parallel, before Subcommittee 21 of ISO's Technical Committee 97 had defined the entire set of services. In theory, it also allows individual service definitions to be modified without disturbing other layers in the model. However, few users are likely to implement protocols in such a way that their interfaces correspond to all the service boundaries. Instead, for instance, a single ROM chip might provide the functions of two or three layers together.

The ISO specifies protocols for each service definition within the layers of the model. These are descriptions of the bit coding formats in which specific information is passed between processes, as well as the procedures to interpret it. Protocols operate between "peer entities"—the parts of a system providing services for a given layer—in the different end systems. Thus information about Network Layer protocols in a message sent by one system is used only by the Network Layer in the receiving system.

A number of protocols may implement a given service, and more than one service may be provided by each layer of the Reference Model. In this respect, OSI can be viewed as a collection of worldwide engineering design activities, with overall coordination provided by the ISO Reference Model. Thousands of engineers from hundreds of organizations worldwide participate, including all major computer and network manufacturers.

The service definitions and protocols are in various states of development for each layer [Fig. 3]. Service definitions have been completed for Layers 2, 3, and 4, and work on other layers is well underway. Protocols for some layers are already international standards, including Network, Transport, and Session layers. The working group's goal is to complete the initial set of protocols for the Application and Presentation layers of the Reference Model by the end of 1986.

Some manufacturers have already moved to implement OSI protocols that have been completed. General Motors, for example, is using one set of OSI protocols in its Manufacturing Automation Protocol (MAP). Boeing is proposing a similar set for its Technical and Office Protocols (TOP). [The MAP and TOP standards will be covered in the April issue of *IEEE Spectrum*.]

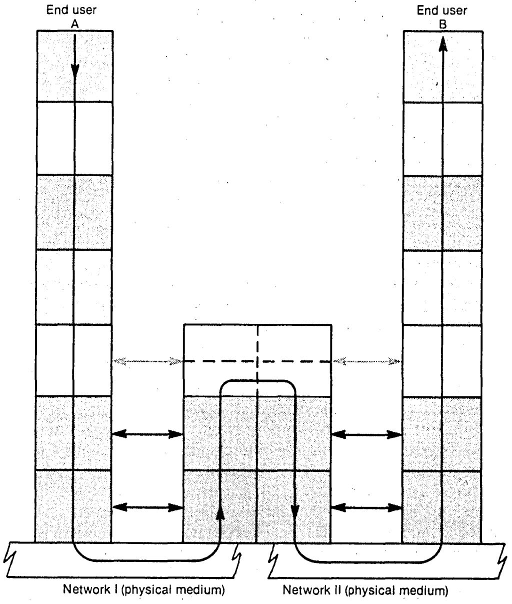
An application using a different set of OSI protocols, however, may or may not be able to communicate with MAP and TOP. OSI does not allow all computers and communications networks to communicate automatically and at will. Rather, OSI users will form "communities of interest" to limit the options. They will define a set of services to be provided by communicating machines in their particular industry and then implement those services in a handful of protocol options for each level.

Lower layers based on existing standards

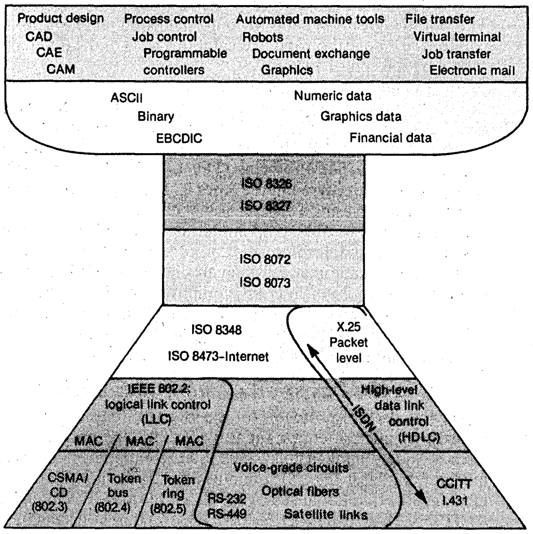
The initial set of protocols for the lower OSI layers are based on existing international standards and thus the protocols are already widely implemented. There are a number of physical medium standards for OSI communication over short distances, including the traditional analog RS-232C, the more recent digital CCITT X.21, and the IEEE 802 LAN standards (ISO 8802.3, 8802.4, and 8802.5). [For a comparison of the IEEE LANs with the OSI model, see "Lining up against the layers," p. 68.]

For longer-distance communication among OSI applications, the physical interface for the Integrated Services Digital Network (ISDN) may become the dominant standard [see "A universal plug already developed," p. 70].

The set of protocols that provides the services of Layer 2 over X.25 networks is called the High-level Data Link Control. Sev-



[2] A message may pass through many relay systems on its way between application processes. In an OSI application, the path taken is invisible to the end users. Only a relay system "knows" what route a message is using.



[3] The "OSI wineglass" of protocols shows the many functions covered by upper layers and the multiple options for physical media at the lower layers. The Session and Transport Layers, however, have fewer alternatives and are now international standards.

eral subsets have been defined; work is proceeding on others. The first to be defined was the CCITT X.25 Link Access Procedure B, for balanced connection-oriented communication—that is, a one-to-one link over a dedicated circuit between two parties.

Next to come was an option allowing multilink communication, or splitting a single communication among several physical channels. And most recently a new subset provides multiplexing functions—allowing several communications to use a single physical channel. Three types of service are provided by the High-level Data Link Control: connection-oriented, connectionless, and single-frame transmission.

Connection-oriented service requires a connection to be established between the two end systems before the communication is transmitted. The connection can be either physical—a set of wires—or "virtual"—preplanned routes over which packets will travel. A good analogy here is a telephone call; a line is established and dedicated to a particular conversation before the two parties begin talking.

Connectionless service involves communication in which each data unit, or packet, travels independently. The path may be established in advance—as on certain LANs—or as the message arrives at each network junction. A good analogy for this type of service is mailing a letter, since it will travel to its destination independently of any others sent to the same address, regardless of whether the same route is used.

Single-frame transmission sends only one frame of data at a time. An example of this is a remote sensor that transmits a signal to a guard station if it detects motion.

Layer 2 protocols may break a stream of data up into frames, which are transmitted sequentially, and may require a frame acknowledgment signal from the receiving system. If so, the frame is retransmitted if the signal is not received. The Data Link Layer may also provide flow control—monitoring the rate of frame transfer—so that systems can exchange data at different speeds.

How to connect networks

As for Layer 3, the Network Layer, its service definition includes network connection, data transfer, reset, and connection-release functions. Expedited data and receipt-confirmation services are optional and are specified when a network connection is established. The receipt-confirmation service supports conformance to the CCITT X.25 standard. An addendum to ISO 8348 is now being developed to add connectionless network service—the simple transfer of a data unit.

The Network Layer must provide for many network types, only some of which have been fully identified. Each type or family of protocols within the layer has a unique identifier, so the protocol can be identified and changed during the transmission of a message. All protocols in the 1984 revision of X.25 are accommodated without change.

The service definition (ISO 8072) and protocol specification (ISO 8073) are now approved for Layer 4, the Transport Layer. ISO 8073 specifies several classes of protocol for connection-oriented communication, with a wide range of functionality—from the simple (Class 0), for use with highly reliable X.25 networks, to high-quality service (Class 4), with error detection and recovery for possibly unreliable networks.

Specifically, Class 4 service ensures that data is not lost, duplicated, or corrupted in transit and that it arrives at its destination in the right order. The Transport Layer can also provide end-to-end error checking between communicating parties, or it may rely on the quality of service provided by the Network Layer. Work is now underway on Transport Layer service definition and protocol specification for connectionless data transmission.

The service definition (ISO 8326) and protocol specification (ISO 8327) are also approved for Layer 5, the Session Layer. While this layer is full of options among the facilities available to the users, initial implementations will contain two subsets of service definitions: the session kernel, for establishing and releasing a session; and the basic combined subset, which adds token management—a request for use of resources—to the kernel.

For Layer 6, the Presentation Layer, an Abstract Syntax Notation 1 developed by CCITT has been adopted as ISO 8824 to provide rules for defining and recording the meaning, or semantic content, of messages. Associated with this are a basic encoding rule (ISO 8825), as well as custom encodings registered with ISO, to turn such notations into actual messages for transfer.

Layer 7, the Application Layer, is the only one that provides services directly to the application process. It does so by drawing on the services of all six layers below it. Conceptually, the Application Layer is broken down into three parts: a user element, common-application service elements (CASEs), and specific-application service elements (SASEs).

The user element represents functions specific to the application process that needs to communicate. It selects among the services offered by the rest of the layer, including the CASEs and SASEs, on behalf of the application program.

The CASEs are general-use capabilities needed by nearly all applications. Included among CASE functions are commitment, concurrency, and recovery for distributed processing.

The SASEs include file transfer, access, and management; job transfer and manipulation, for distributed batch jobs; message handling facilities; virtual terminal systems, which allow remote systems to communicate as terminals; and directory services.

Serving 'communities of interest'

These functions serve specific industries, known as communities of interest. The financial services and banking industry is one example of a broad community of interest; another is the users of automated industrial equipment. Each group has unique needs and requires Application Layer services specific to the industry.

For example, industrial automation applications may not have a high volume of on-line inquiry. But because factory communication must occur in real time—as opposed to sending messages in batches—the maximum permissible waiting time between commands sent to the machine tools must be very low. In this environment, real-time "foreground" protocols will be carefully designed for high performance. For "background" processing—analyzing production data, for instance—the job transfer and manipulation function of the SASE might be used.

The message from the Application Process, plus information added by each layer below, forms the frame that is sent out over the network [Fig. 4]. At each layer, header control information is appended to the data unit received from the layer above. This information identifies the protocol options used and gives other data about the message and its routing. At the receiving end, the header information is removed and processed by each layer. Then the remaining data unit is passed up to the next layer, where a similar operation takes place.

A good analogy to show how the functions of the OSI model operate is the production and transmission of a simple business letter. It parallels the OSI process, using only the language of business communications; no computer terminology is needed [Fig. 5].

The thank-you letter

Imagine that the president of a West German company has agreed to buy 50 tons of wheat from a firm in Wichita, Kan. Because he got a good price, he asks the public relations manager to send a thank-you note to the sales director of the Wichita firm.

The West German executive represents the application process that initiates a communication. He deals in terms of the meaning of the communication, or the semantics; he merely tells the PR manager to send a thank-you note. The PR manager actually gets the machinery going. He is the Specific Application Service Element of the Application Layer, calling on the services of the layers below him to meet his needs in transmitting the message.

The West German PR manager dictates the note onto a cassette tape and gives it to his secretary—who acts as the Presentation Layer. She translates the message into English and types it as a formal business letter. In OSI terms, she has prepared the Transfer Syntax—a string of data in a language common to the sender and the receiver—in this case, English.

After typing it, the secretary hands the letter to her administrative assistant—the Session Layer. He records the letter in the German company's file on Wichita Wheat Co., ensuring that the right person has been addressed, with the correct title and spelling, exact office number, and other details. This checking allows both ends of the communication to organize and synchronize their dialogue, by noting where the message goes and when it was sent. If there is back-and-forth exchange of information, the Session Layer will manage the dialogue.

The next layer—Transport—is provided by the manager of shipping and receiving. His job is to negotiate the quality of service available from the Network Layer, approve the connection, and provide receipt and delivery. He is really guaranteeing end-to-end transmission. If something untoward happens during transmission, he will recover by sending another copy of the letter—hence he always copies a letter before sending it.

After copying the letter, he assigns a sequence number (in this case, "1 of 1"). Then he passes the shipment—tagged with both destination address and phone sequence number—to a shipping clerk. He tells the clerk to establish a route over which the note will be sent to Kansas. The Network Layer (the shipping clerk) will select the routing and advise the Transport Layer (the transport manager) of it.

The shipping clerk calls his counterpart in the German company's New York City office. He learns that the company's internal mail service can take the shipment to the New York office, and Federal Express will deliver it to Wichita the next day. Note that OSI applies to communications over private networks (the company's internal mail operation) and public networks (Federal Express).

He attaches a routing slip and puts the letter with others into a mail cart labeled "New York." Then he sends the cart to the mailroom, which serves as the Data Link Layer.

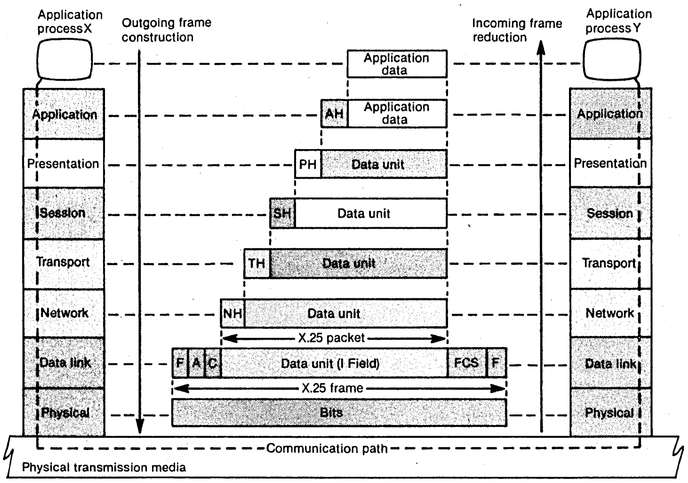
The mailroom workers also make copies of everything they receive, bag the mail, and weigh it on a very accurate scale. They note the destination and weight of each mailbag on a tag attached to the bag. Then they move the bag to the loading dock—the Physical Layer, or the interface to the physical medium (the trucks, trains, and airplanes to take it to the United States).

The workers on the dock call the trucks and load the mailbags onto them when they arrive. At this point, the "bits" have left the machine and are in transit on the medium—the communication has been sent on its journey.

When the mailbag arrives in New York City, the workers on the New York loading dock—the Physical Layer—pass the mailbag to the workers in their mailroom—the Data Link Layer. This mailroom has a scale identical to the one in West Germany, which can detect the loss of even one letter from the mailbag. If the weight of the bag does not match that on the label, the whole shipment is rejected and the mailroom in Germany is notified to send replacement copies of all the letters, using the duplicates they have kept.

This task represents "frame check sequences" performed by the Data Link Layer. In this case, the weight of the letters matches exactly, so the New York mailroom sends word back to Germany that the mailbag is OK. Then the shipment goes to the routing clerk in New York—the Network Layer—who opens the mailbag and sorts the mail.

Mail for employees in the New York office gets passed along to the transport manager—the Transport Layer—for processing up



[4] A message passed from Application Process "X" down through the layers to an X.25 network acquires header information from the functions of each layer. The receiving Application Process "Y" does not see this, however; its Application Layer passes along only the message sent by "X." The message is stripped of all its headers and frame information by the layers below the application process. The bitstream actually sent over the network is an X.25 data frame.

in the organization. Other mail remains at the Network Layer to be rerouted. The routing clerk recognizes the thank-you letter as one to be sent through Federal Express, so she tags it for Federal Express and sents it back to the mailroom.

The mailroom groups together (multiplexes) all mail for Federal Express delivery to the Wichita firm, as there are many letters concerning the grain deal. Again the contents are copied, weighed, sealed (in a Federal Express package), and tagged with a new shipment number and address. The bags go out onto the loading dock and away in the Federal Express trucks.

Assuming Federal Express and the Wichita firm use an OSI model, they will go through a similar process to route the package. In all the cases, only the lower three layers—Network, Data Link, and Physical—are involved when a message is routed via intermediate networks. The upper layers — Transport and above—are involved only at the origination and destination of a communication.

When the Federal Express package arrives in Wichita, the routing clerk passes it up to the transport manager, who checks the packing slip and telephones her counterpart in Germany to let him know that the letter has arrived in good order.

In this way the Transport Layer acknowledges "end to end" communications. All previous acknowledgments have been at the Data Link Layer, from one leg of a journey back to the previous leg. This final acknowledgment connects the end of the journey to the beginning, no matter what carriers—reliable or not—have been used in between.

Once the communication has been received and acknowledged by the Transport Layer, it is passed along to the Session Layer. A file clerk logs the letter in the file for the German wheat buyer and takes the letter to the Presentation Layer — the sales director's secretary. She reads the letter and determines that it is in English; no translation from German is necessary.

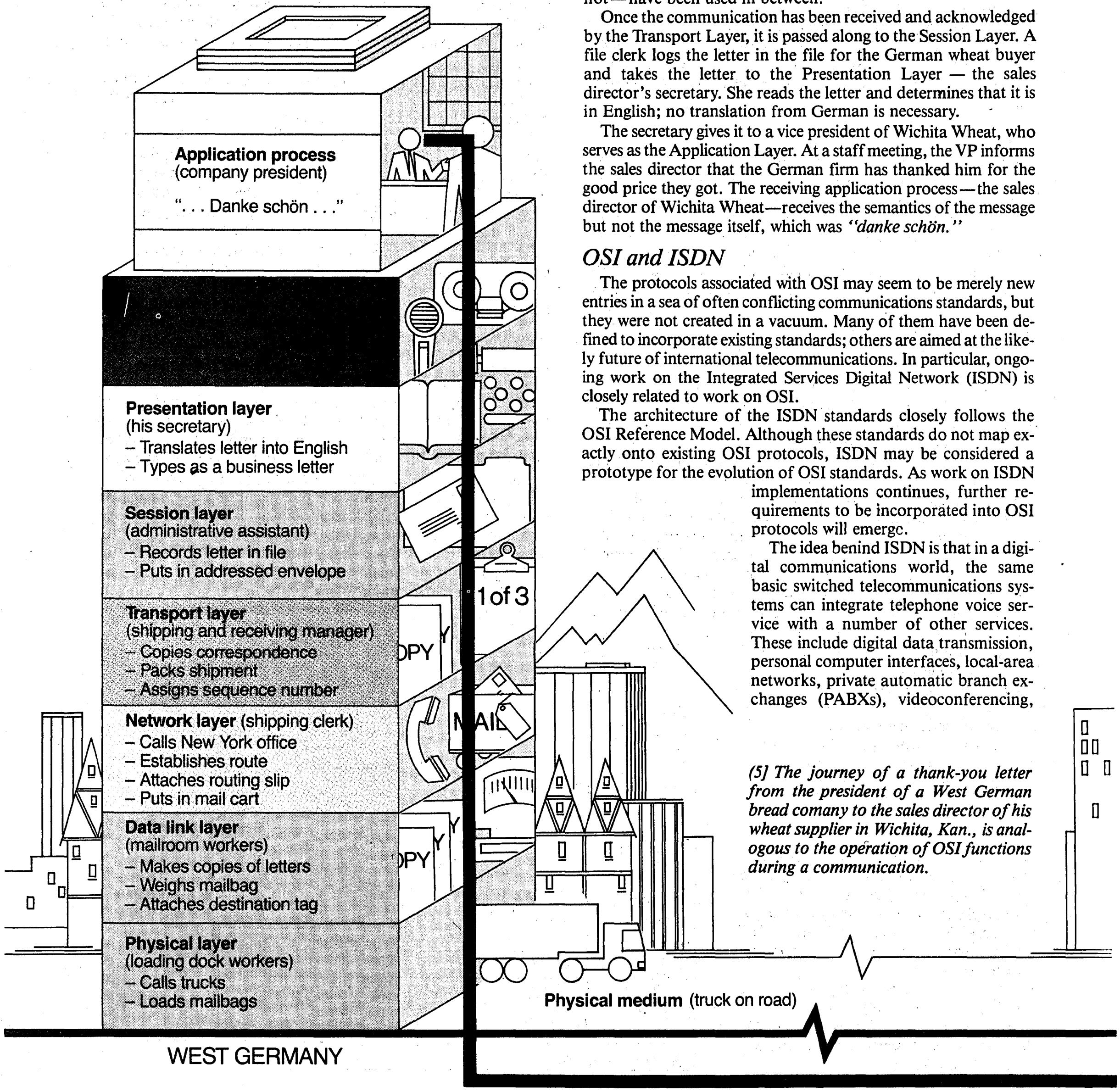
The secretary gives it to a vice president of Wichita Wheat, who serves as the Application Layer. At a staff meeting, the VP informs the sales director that the German firm has thanked him for the good price they got. The receiving application process—the sales director of Wichita Wheat—receives the semantics of the message but not the message itself, which was *"danke schön."*

OSI and ISDN

The protocols associated with OSI may seem to be merely new entries in a sea of often conflicting communications standards, but they were not created in a vacuum. Many of them have been defined to incorporate existing standards; others are aimed at the likely future of international telecommunications. In particular, ongoing work on the Integrated Services Digital Network (ISDN) is closely related to work on OSI.

The architecture of the ISDN standards closely follows the OSI Reference Model. Although these standards do not map exactly onto existing OSI protocols, ISDN may be considered a prototype for the evolution of OSI standards. As work on ISDN implementations continues, further requirements to be incorporated into OSI protocols will emerge.

The idea benind ISDN is that in a digital communications world, the same basic switched telecommunications systems can integrate telephone voice service with a number of other services. These include digital data transmission, personal computer interfaces, local-area networks, private automatic branch exchanges (PABXs), videoconferencing,



[5] The journey of a thank-you letter from the president of a West German bread comany to the sales director of his wheat supplier in Wichita, Kan., is analogous to the operation of OSI functions during a communication.

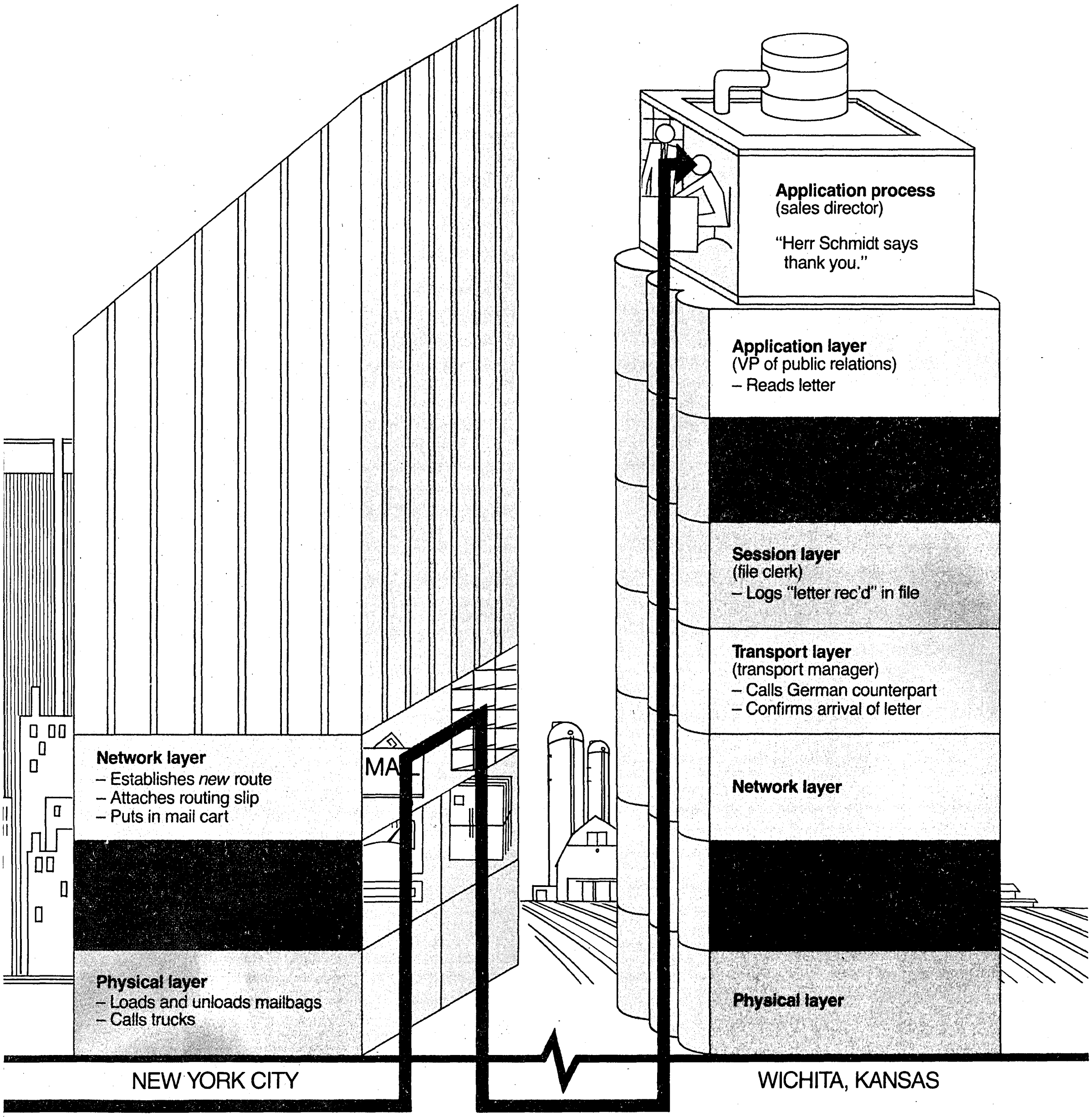
and joint-use remote applications, like automatic teller machines and self-service fuel pumps.

Only the lower three layers of OSI are applicable to initial ISDN work [Fig. 6]. The basic ISDN interface is composed of a 16-kilobit-per-second signaling channel (D-channel) plus two circuit-switched 64-kb/s digital channels (B-channels). Depending on the characteristics of connecting networks, ISDN offers access to the D-channel alone or in combination with one or both B-channels.

In the basic service, the ISDN Physical Layer operates with a bit stream of 192 kb/s and provides a multiplexing arrangement. Of this, 48 kb/s is control information that facilitates the multiplexing.

The signaling channel uses the Link Access Procedure D protocol for Data Link Layer services. It provides multiplexing for three functions: signaling information that controls switching connections on the B-channels; low-speed packet-switched services; and optional channels that can be used for sporadic low-bandwidth transmission, like burglar-alarm signaling.

The OSI Network Layer protocol for the D-channel is specified by CCITT recommendation Q.931. It provides the mechanism for making and breaking connections on the



Lining up against the layers

To understand the division of functions among the layers of the Open Systems Interconnection (OSI) model, it is helpful to compare it with some existing networks and communications standards. Perhaps the most widely known among data processing professionals is IBM's Systems Network Architecture (SNA), designed to provide a common architecture for communication within the company's diverse equipment.

The distribution of functionality among the layers of SNA varies significantly from that of OSI; the number of options is not as large, because SNA is intended for a limited set of machines operating in a known network environment.

The SNA transaction services layer has elements of both the application process and the OSI Application Layer. It provides services necessary for operation of the network—for instance, a program to agree on the number of sessions between network nodes. But it also includes such functions as electronic mail, defined under OSI as an application process.

The presentation services layer is comparable to the rest of the OSI Application Layer and to the entire Presentation Layer. It provides a high-level interface to application programs, taking high-level statements—such as SEND DATA—and translating them into lower-level service requests.

One "half-session" serves each user on either end of a communication, although more than one user may use a piece of equipment on the network. A half-session includes resources for data flow control — controlling requests and responses in a dialogue — and transmission control, managing buffer resources and expediting data. The services manager responds to requests from the presentation services layer to create, assign, and destroy conversations.

Beneath the half-session, the path control function is divided into three layers for IBM's "backbone" machines, predominantly mainframe computers. Peripheral equipment —personal computers, minicomputers, and terminals—uses a much simpler path control function without the sublayers.

Virtual route control provides flow control and error control to the logical—as opposed to the physical—network. Explicit route control establishes physical paths for connection. Transmission group control provides a multilink capability and ensures data delivery. And the SNA data link control provides reliable transfer of information between nodes, as does the OSI Data Link Layer. The physical layer is not explicitly defined in SNA, since the medium is known, but the OSI Physical Layer functions are implicit in the architecture.

The ARPAnet — developed for research purposes by the Department of Defense in the early 1970s — has a different distribution of functionality. Here too, the Application Layer provides the same user services as does the OSI equivalent. But there is no ARPAnet equivalent of the OSI Presentation and Session layers; either ASCII characters or data bits are passed through the network, and the communicating applications know what to send and expect.

The ARPAnet Service Layer provides OSI Transport Layer functions and a few from the Session Layer — namely the ability of the network itself to initiate a "graceful close" to a communication session if a user drops off the network.

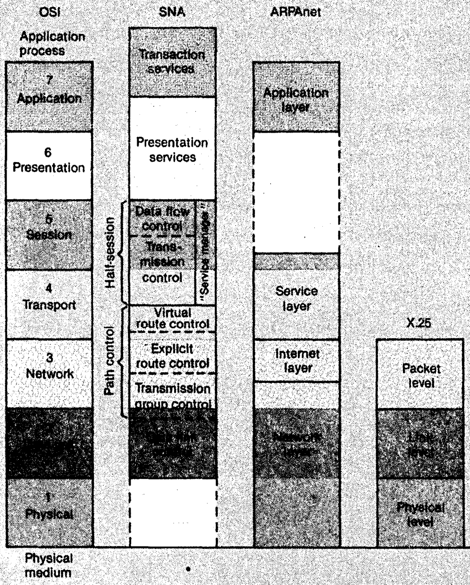
ARPAnet's Internet Layer routes communications among networks, either to intermediate networks or to an application process if the message has reached its destination. Finally, the rest of OSI's Network Layer functions and the Data Link and Physical Layers are contained in ARPAnet's Network Layer. This layer addresses the real characteristics of the many actual networks across which the ARPAnet system operates. The functions are not strictly independent of the physical medium, however, as is the OSI Physical Layer.

The X.25 standard for packet-switched communication over national and international data networks defines one major way to implement network services.

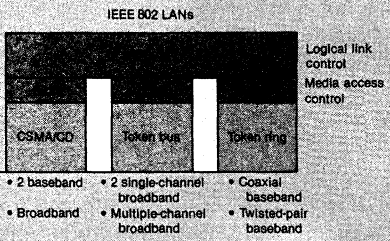
The functionality of the latest X.25 specification, issued in 1984, corresponds entirely to OSI. The X.25 Packet Level protocol, which routes and switches packets through network nodes, corresponds to the OSI Network Layer. The X.25 Link Level guarantees reliable data transfer across the physical link—as does the OSI Data Link Layer. And the Physical Level of X.25 includes the functional and procedural characteristics to activate, maintain, and deactivate the physical connection.

The IEEE 802 local-area networks (LANs), which define three commonly used networks for single-site communications, also conform to OSI. The 802.2 Link Level Control (LLC) procedures compare to the upper half of the OSI Data Link Layer, and the lower half is matched by the Medium Access Control (MAC) functions of each individual LAN standard.

Depending on the implementation, the LLC may provide such services as end-to-end error control, flow control, and sequencing. Beneath the LLC, each LAN type has its own MAC standard to provide data requests, confirmation of data requests, and data transfer services. —J.V.



Compared with the Open Systems Interconnection (OSI) model, architectures like IBM's Systems Network Architecture (SNA) and communications networks like the ARPAnet perform most of the same functions but divide them differently. The international X.25 standard for long-distance voice and data transfer provides only Network Layer functions and below; the IEEE 802 LAN standards provide only Data Link and Physical Layer functions.



B-channels and for other ISDN control functions. For the packet-switching function, Layer 3 is the packet level of X.25. The protocols for optional functions will be defined by the CCITT at a later date or may be specified as a national option.

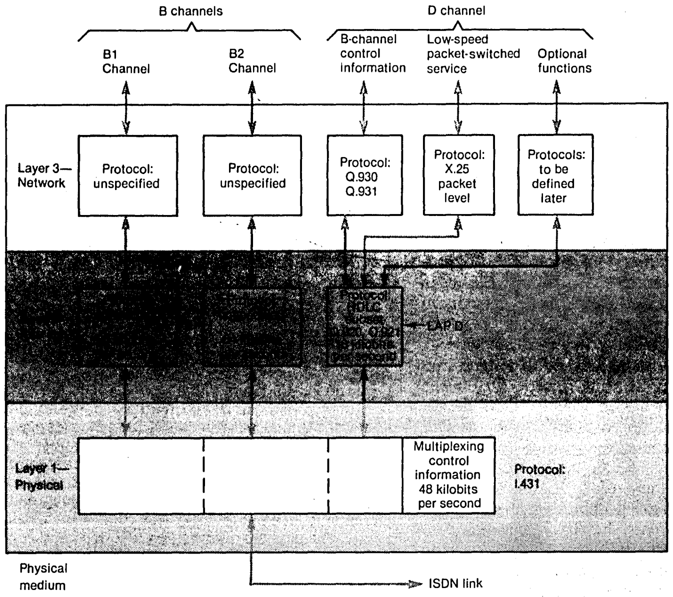
The data link and network protocols are unspecified for the B-channels; these channels provide a "transparent" facility that may use whatever protocols are appropriate for the application.

Above the Network Layer, ISDN protocols depend on the application being used. CCITT Recommendation I.212 covers the upper four layers of ISDN services, referring to them as Teleservices. Protocol recommendations for Layers 4 through 7 have been developed by CCITT.

But will it work?

Computer users—many stung in the past by false promises of compatibility—may be inclined to greet claims of compatibility with skepticism. If OSI is to catch on, there must be a way to verify that products conform to its definitions. Work on testing products for OSI conformance has just begun but is developing rapidly.

The main influence in the United States to date has been the National Bureau of Standards, headquartered in Gaithersburg, Md. A newly formed group of equipment manufacturers, the Corporation for Open Systems in Washington, D.C., is also likely to become an important factor in OSI testing.



B channels
D channel
B1 Channel
B2 Channel
B-channel control information
Low-speed packet-switched service
Optional functions
Layer 3— Network
Protocol: unspecified
Protocol: unspecified
Protocol: Q.930 Q.931
Protocol: X.25 packet level
Protocols: to be defined later
Protocol HDLC subset Q.920 Q.921 16 kilobits per second
LAP D
Layer 1— Physical
Multiplexing control information 48 kilobits per second
Protocol: I.431
Physical medium
ISDN link

[6] Initial ISDN work is concentrated in the Physical, Data Link, and Network layers of OSI. Above the network services, protocols for ISDN will depend on the application. For example, teletex terminal equipment interface specifications, character sets, and mixed-mode terminal capabilities are included in the OSI Application Layer protocols.

The NBS does not provide testing services to manufacturers; it simply develops methods and software to test conformance to various OSI protocols and sells them through the National Technical Information Service. Currently, software to test the complete set of protocols for the OSI Transport Layer and the upper—or Internet—portion of the Network Layer are available.

Under development at NBS are tests for the messaging, file transfer, and virtual-terminal protocols for the Application Layer; the agency is developing these test programs under a Department of Defense contract. Also being developed are programs to test the Physical Layer and the bottom half of the Data Link Layer for the IEEE token-bus local-area network standard.

The NBS hopes to bring on line shortly a service called Osinet, a nationwide network for manufacturers interested in OSI testing. There are three immediate goals, according to John F. Heafner, chief of the systems and network architecture division at the NBS: to promote the development and dissemination of testing systems; to allow vendor-to-vendor testing of products, or interoperability testing; and to offer demonstrations of OSI testing services and products.

A more distant goal is to tie together OSI testing centers around the world, Heafner said, noting: "We would like to be able to offer testing for worldwide product conformance, but only if we can be assured that there will be no trade barriers created to protect individual markets."

The goals of the Corporation for Open Systems are equally ambitious. Initiated by the Computer and Communications Industry Association (CCIA), the corporation will encourage the development of test capabilities that manufacturers can use during product development, to reassure customers that products being marketed conform to the appropriate standards.

The corporation, formed late last year by a group of 18 suppliers of computer and communications equipment, is not a standard-setting body, noted Jack Biddle, president of the CCIA. But it does intend to promote development of a universally accepted set of OSI protocols for individual applications, including such standard data processing functions as file transfer and management and electronic message handling.

The group does not plan to provide testing services, Biddle said, but instead will develop testing programs and services or subcontract this development to others. A possible provider of those services, he noted, is the Industrial Technology Institute of Ann Arbor, Mich. This not-for-profit organization is currently involved in testing compatibility with the MAP specification among suppliers of factory automation equipment.

Over the long term, the corporation hopes to convince executives in the computer and communications industries of the strategic importance of a single open network architecture. "There is a need for greater voluntary efforts in the standards community," Biddle said. "These activities are not yet accepted as an integral part of product planning strategies by many companies."

Biddle is confident that testing for OSI conformance will become vitally important to the world electronics market, but he expects "a long struggle to make it happen." Asked whether users were really demanding OSI-compatible equipment, he said: "You should have heard my phone ringing off the hook after the word got out about the corporation. They are frustrated, they don't like buying from just one vendor, and they want solutions."

IBM pursuing OSI

A big question is how OSI will affect IBM. For a quarter of a century, IBM has been the leader in the computer industry. Its 11-year-old Systems Network Architecture (SNA) is the most widely implemented communications architecture for mainframe computers, and IBM has often functioned as a de facto standard-setting body for computer networking.

At the start of work to define the OSI Reference Model in the late 1970s, IBM participated in standards meetings and technical sessions. "IBM contributed very significantly and very constructively," said Harold Folts, president of Omnicom Inc. in Vienna, Va., a telecommunications consulting and education concern.

"And there is no question that they are moving in the direction of OSI for the European market."

Last year Digital Equipment Corp. announced that it would gradually modify its Digital Network Architecture to conform to OSI protocols as they are developed. The 12 major European computer manufacturers have indicated that they too will adopt OSI protocols. To promote OSI, some European governments have introduced regulations requiring OSI compatibility in new data network installations. With a significant presence in Europe — as indeed it has anywhere — IBM has announced plans to support OSI there.

The company stated: "As standards for Layers 6 and 7 are agreed upon over the next two years, based on business considerations, IBM will develop products that will meet the requirements of both the customers and OSI standards." IBM said that "OSI will complement the well-proven SNA architecture" and that "OSI and SNA can supplement each other to provide a balanced solution for the management of networks and for the transfer of information between them."

IBM Europe offers OSI capability through Layer 5, which indicates that the company will offer OSI implementations in addition to its own SNA architecture. [For a comparison of SNA with the layers of OSI, see "Lining up against the layers," p. 68.] The company's center for research on OSI implementations is the IBM European Networking Center in Heidelberg, West Germany.

IBM's Open-systems Transport and Session Support software, first shipped last December, supports most functions of OSI Layers 4 and 5 on the IBM/370 mainframe. The company has also offered several products for Layers 1 to 3 of OSI, mainly interfaces for various equipment to connect to X.21 and X.25 communications networks. But the Open-systems Transport and Session Support software is IBM's first comprehensive offering for OSI connectability above the Network Layer.

Currently the company is testing the X.400 messaging standard, a set of CCITT recommendations developed within the OSI framework. IBM may attempt to provide a bridge between its own document architectures and X.400. There will ultimately be a host of applications to which IBM's massive array of equipment will have to be connected, including electronic mail, teletex, videotex, and other such European services.

Many observers feel that ultimately IBM will offer not only full OSI implementations but also gateways to allow OSI to interconnect with existing SNA networks. "There will be a migration of SNA to OSI standards, probably without a lot of flag waving," said Folts of Omnicom. "They will offer two standards to start with, then they will merge—they can afford to make major leaps without worrying about backward compatibility." IBM has also recently joined the Corporation for Open Systems.

The implications are profound. If IBM's equipment uses essentially the same communications protocols as those of its many competitors, the company will be forced to compete increasingly on the technological merits of its products and perhaps on price. In a sense IBM will lose some control over the direction of computer equipment and design that it has enjoyed — particularly in the United States — for the last quarter of a century.

In some ways the promise of OSI has been oversold. It is not a magic cure-all that will allow every variety of computer equipment to be plugged together as stereo components are.

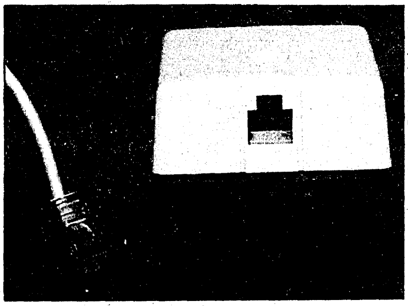
But OSI probably has a better chance than most of living up to its potential. For one thing, the group of potential users for OSI implementations spans many countries and diverse industries. Many suppliers will compete to supply conforming equipment.

OSI users can also decide which protocols are appropriate for their own needs. The best examples so far are the MAP and TOP standards, and there will be more as OSI gains public attention. The banking community, for instance, is working hard to apply OSI to electronic funds transfer and other services.

Finally, OSI leaves room for inevitable growth and change in a most elegant way. Standardizing protocols between functions — but not the design for implementing those functions — ensures compatibility between different systems while leaving room for innovative engineering.

One communications design engineer told *Spectrum* that "the only interesting question provoked by OSI is whether we end up with communications provided by the computer industry, or computers made by the communications industry." The answer may not be clear for decades. But OSI will provide a giant step toward the worldwide integration of computing and communication. From any perspective, Open Systems Interconnection promises to affect every part of both industries. It is, in the words of the same designer, "the only game in town."

A universal plug already developed


AT&T-IS Consumer Products Laboratory

The RJ-45 minimodular connector is likely to be approved as the universal interface for the Integrated Services Digital Network (ISDN), and thus ideally would be used for many Open Systems Interconnection (OSI) applications. Developed by AT&T Bell Laboratories, it is an eight-wire version of the familiar RJ-11 jack and plug widely used in U.S. telephone terminals and instruments (see photo).

The pins are arranged as follows: 1 and 2 are power sources, 3 and 6 transmit, 4 and 5 receive, and 7 and 8 are power sinks. Because the plug centers itself in the socket, the current four-pin plug would contact pins 3 to 6, allowing customer premises in the United States to be wired with the new socket and still remain compatible with existing telephone equipment.

While it may appear to be fragile, the plug has proved to be quite rugged, and it meets a number of criteria: it is small, keyed, self-orienting, and can be released without any tools. The connector set is now an International Organization for Standardization draft standard, and chances for final approval appear good.

—*J.V.*

To probe further

Over 20 articles in the *Proceedings of the IEEE* for December 1983 cover virtually all aspects of Open Systems Interconnection in detail. This issue can be ordered from the IEEE Service Center, 445 Hoes Lane, Piscataway, N.J. 08854.

An index of standards relating to OSI is available from Omnicom Inc., 501 Church St. NE, Suite 304, Vienna, Va. 22180. Proposed, draft, and approved ISO standards are available from the American National Standards Institute, 1430 Broadway, New York, N.Y. 10018.

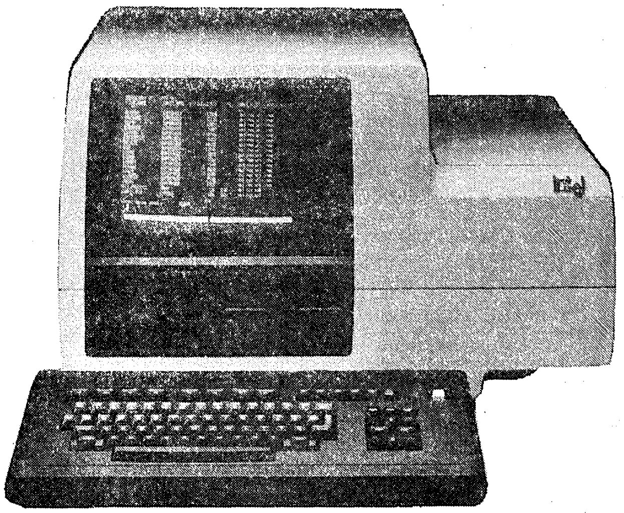
The IEEE 802 LAN standard documents (802.2, 802.3, 802.4, and 802.5) are available from the IEEE Service Center. For further discussion of IEEE 802 LANs, see "Local area nets: a pair of standards," by Maris Graube in the June 1982 issue of *Spectrum*. For more details on ISDN, see "The innovation revolution awaits," by Paul Wallich and Glenn Zorpette, in the Nov. 1985 issue of *Spectrum*. Copies of both issues are available from the IEEE Service Center. ◆

Microcomputer Development Systems 5

iMDX 430/431/440/441 INTELLEC® SERIES IV MICROCOMPUTER DEVELOPMENT SYSTEM

- **Complete Microcomputer Development System for the iAPX 86/87/88/186/188/286, the MCS® -80/85 and the MCS -48/51/96 Family Microprocessors**
- **Advanced, Friendly Human Interface with Menu-Driven Function Keys, HELP, and Syntax Builder/Checker Capabilities for Increased User Productivity**
- **Foreground/Background Multiprocessing for Simultaneous Execution of Two Jobs by a Single User; Increasing System Throughput**
- **Multi-User Capability for Simultaneous Operation by Two Users, Significantly Reducing System Cost per User**
- **Hierarchical File System Provides File Sharing and Protection for Large Software Projects**
- **Software Compatible with Both Series IIE and Series IIIE Development Systems**
- **Supports PL/M, Pascal, C, and FORTRAN, and Basic High-Level Languages as well as Assemblers**
- **Provides Program Management Tools (PMTs), Advanced AEDIT Text Editor and Supports Powerful PSCOPE Symbolic, Source Level Debugger**
- **Can be Fully Integrated into the NDS-II Network Development System**

The Intellec® Series IV is a new generation development system specifically designed for supporting the iAPX family of advanced microprocessors. It also supports the MCS-80/85 and the MCS-48/51 families.



230625-1

Figure 1. Intellec® Series IV Microcomputer Development System

October 1986
Order Number: 230625-005

Series IV provides a state-of-the-art, easy-to-use, high performance host environment for running a wide variety of hardware and software development tools. A unique combination of tools provides an integrated microcomputer system design that results in highly improved designer productivity and considerable shortening of time to market. The length of the compile-link-load-debug-edit cycle is minimized by the friendly human interface, powerful and easy-to-use editors, a wide selection of language translators, source level debuggers, program management tools. The advanced operating system features a hierarchical file system, foreground/background multitasking, and multi-user capability. Furthermore, the Series IV can serve as a powerful workstation on the NDS-II distributed processing network for high performance milti-user software development. The networking architecture supports a distributed co-operative processing environment. Tasks like compilations can be executed in the background mode or exported to an idle workstation while the user is in the middle of an interactive edit session. The key benefit of this approach is a much higher system throughput and programmer productivity than, for instance, a system designed for raw-performance and fast compilations only.

The Series IV is offered in four different versions, providing a range of storage and performance options so that the user may select the configuration to suit his/her stand-alone or networking development station needs. The four versions are not only compatible with one another, but are also software compatible with the current generation enhanced Series IIE/IIIE systems. Existing ISIS-compatible software can run directly on the Series IV under the ISIS operation system. Finally, the NDS-II network provides an ideal means for the various hosts, e.g., Series II/III/IV to work with each other, protecting the user's past, and present, and future investment.

FUNCTIONAL DESCRIPTION

Systems Components

The Intellec Series IV model 430/431 Microcomputer Development System is an easy-to-use high-performance system in one package. It includes a CPU board for each of the iAPX 88 and MCS 85 processors and 640K bytes of system RAM. The system has eight function keys included in its detachable standard ASCII keyboard that also has cursor controls and uppercase/lowercase capability.

These function keys are menu driven and, with the use of the syntax builder/checker, greatly reduce user keystrokes. Peripheral configurations include: Model iMDX430WD, 440WD—two floppy disks, one 35MB Winchester; and Model iMDX 431, 441— one floppy disk, one 10MB Winchester.

The 5.25" drives, a green phosphor screen, and a detachable keyboard are all integrated into the system. The main chassis has ten MULTIBUS® slots (three 12" × 12", seven 6¾" × 12") power supplies, fans and cables.

Operating System Environments/ Features

The Series IV provides both an 8086/8088-based development environment and an 8080/8085 based development environment. The host execution mode is the 8086/8088, which runs under the iNDX operating system. To execute an 8080/8085 program, the ISIS-IV utility is invoked, entering the 8085 execution mode. All ISIS-compatible 8-bit software can thus be run directly on the Series IV, through a user interface that is compatible with ISIS-based development systems such as the Series II and the Series III.

HIERARCHICAL FILE SYSTEM

The iNDX operating system employs a hierarchical file system, providing file sharing and protection features. The hierarchical structure allows logical grouping of data. The structure resembles an inverted tree. The root of the system is called the logical system root. The system root logically "connects" the volumes within the file system. Each volume corresponds to a physical mass storage device. Volumes are further divided into files. Files can be either directory files or data files. Directory files contain references to further directory or data files. Data files contain only data.

It is not necessary to know the physical location of files to address them. Each file can be addressed by a path name, which is a character string recognized by the operating system.

The iNDX file system provides file protection features in the form of access rights. The owners of a file may set their access rights to their own files and separately set the WORLD's access rights (everyone else) to their files. File may thus be shared and also protected from accidental or deliberate addressing or destruction.

SINGLE-USER FOREGROUND/ BACKGROUND PROCESSING

Foreground/background processing capability allows the simultaneous execution of two jobs, resulting in improved system throughput. While a program is executing in the background, another program could be run in the foreground. For example, an interactive editor could be executing in the foreground while a compilation is taking place in the background.

A toggle key on the Series IV keyboard can be used to instantaneously move from one region to the other, allowing interactive operations in both foreground and background regions. For example, while a software debug session is taking place in the foreground, listing files can be displayed from the background.

MULTI-USER CAPABILITY

A low cost terminal can be attached to serial port 1. This terminal operates as an independent system, accessing one region, while the console and keyboard access the other region. In this mode two users will be able to perform software development tasks simultaneously at a significantly reduced cost per user.

The Human Interface

The Series IV is one of the easiest systems to learn and to use, as its human interface is designed to be friendly to both novice and expert users.

It offers eight softkeys that cut the number of keystrokes required to perform a function. On-line HELP provides instantaneous access to command definition. The menu-driven screen interface allows the user to see where he/she is at and to select the next operation. In conjunction with the soft function keys, it allows single key command invocation. The syntax builder and checker completes commands and insures proper command syntax before execution. Features such as type-ahead, auto-repeat keys, and quick view file facility are some of the many other human interface factors that improve programmer productivity.

The AEDIT Text Editor

The AEDIT text editor is one of the most poweful and easy-to-use editors available. It runs under the iNDX opeating system and offers features such as:

- Display and scroll text on the screen
- Move to any character position in the text file or to any point on the screen instantly
- Correct typing mistakes as you type
- Rewrite text by typing new characters over old ones
- Make insertions and deletions easily at any point in a file
- Find any string of characters and substitute another string, querying the operator if desired
- Move or copy sections of text within a file or to/from another file
- Create macros to execute several commands at once, thereby simplifying repetitive editing tasks
- Edit two files simultaneously
- Indent text and delimit long lines automatically
- View lines over 80 characters long

Languages and Utilities

The Series IV supports popular high-level languages such as PL/M, Pascal, FORTRAN, and C, as well as powerful "high-level' macro assemblers such as ASM86. In addition, iRMX™ utilities such as ICU-86, PATCH utility, Files Utility, Crash analyzer and SDM 86 System Debug Monitor are supported by the Series IV.

The high-level language compilers produce code for the target processors. They also contain runtime floating-point arithmetic support for the 8087 Numeric Data Processor.

PSCOPE, the High-Level Language Debugger

The Series IV supports the PSCOPE debugger, an interactive, symbolic debugger for FORTRAN, Pascal, and PL/M programs. Operations are performed on source statements, procedure entry points, labels, and variables, as opposed to machine instructions memory addresses. PSCOPE improves productivity in the debug phase of development and produces more reliable software. It allows the user to peform extensive tests and consistency checks on the programs, and it automates much of the testing.

In-Circuit Emulators

The Series IV supports a host of ICE modules including the powerful I^{2}ICE™ for iAPX family-based development. These tools allow the debugging of microcomputer system hardware and software concurrently, saving considerable development cost and time.

Network Capability

The Series IV may be used as a high-performance workstation for use on the NDS-II Network Development System. It has complete access to all the network resources and facilities on the NDS-II. A stand-alone Series IV can be upgraded to an NDS-II workstation with the addition of an Ethernet Communication Board Set. The background partition of the Series IV may be made available as a network resource.

When configured as an NDS-II workstation, the Series IV can also serve as a host for up to four iMDX-580 ISIS cluster boards, providing a cost effective means for supporting incremental 8-bit software workstations on the network.

System Configurations

Series IV Systems are available in 110V, 60 Hz; 220V and 100V, 50 Hz models.

STAND-ALONE

iMDX 431
Stand-alone Intellec Development system with detachable keyboard and integral green CRT. Included in the main chassis is one 5.25″ floppy and one 5.25″ 10 MB Winchester drive.

iMDX 441 Kit
The same configuration as the iMDX 431, this model has an additional higher performance 8086 CPU.

NETWORK

iMDX 430WS Kit
A two floppy workstation that includes Ethernet NDS-II boards for network operation.

iMDX 440WS Kit
The same configuration as the iMDX 430WS, this system includes a high-performance option for resident 8086 execution and faster performance.

iMDX 430 TO 440 UPGRADE

iMDX 434
High-performance add-on option. Converts a model iMDX 430 or iMDX 431 to a model iMDX 440 or iMDX 441.

NETWORK UPGRADE

iMDX 456
Communication board set converts any Series IV stand-alone system to an NDS II workstation.

ND2TLB

The NDSII/Series IV Toolbox is a software only product that contains a valuable collection of tools developed for the NDSII and SIV user. These tools have been designed to make hybrid development system environments work together and to move fully automate the software developer's task. Many tools are provided with source to allow the engineer to customize these products to their own environment.

SECOND-USER TERMINALS

The following terminals have been tested and found to be interface-compatible with the Series IV CPIO board and can be used as second-user terminals.

LEAR SEIGLER, Model ADM 3A
TELEVIDEO, Model 910+

The following terminals have been successfully tested for interface-compatibility, however they do not meet Intel environmental specifications: adverse electrostatic conditions may produce unpredictable screen output, requiring terminal or system reset.

Televideo, Model 925, 950
Adds Viewpoint 3A+
Qume 102
Hazeltine 1510

PHYSICAL CHARACTERISTICS

Chassis		Keyboard
Width	26.5″ (67.3 cm)	20.0″ (50.8 cm)
Height	16.5″ (41.9 cm)	3.0″ (7.6 cm)
Depth	18.5″ (47.0 cm)	8.0″ (20.3 cm)
Weight	51 lb. (23.4 kg)	7 lb. (3.1 kg)

ELECTRICAL CHARACTERISTICS

DC Power Supplies

Volts Supplied	Amps Supplied
+5.1 ± 1%	45.0
+12 ± 5%	3.0
−12 ± 5%	2.0
−10 ± 5%	0.5
+12 ± 5%	5.0

AC Requirements

110V, 60 Hz
220V, 50 Hz

Environmental Characteristics

Operating Temperature: 10°C to 35°C (50°F to 95°F)
Humidity: 10%–95% (non-condensing)

Equipment Supplied

Series IV System

Series II/III to Series IV link software diskettes and cable

Series IV Software
- iNDX OS
- ISIS IV OS
- AEDIT
- Macroassemblers and utilities
- ICE™ software
- Prom Programmer Software
- Debug 88
- Program Management Tools (MAKE, SVCS)
- Diagnostics

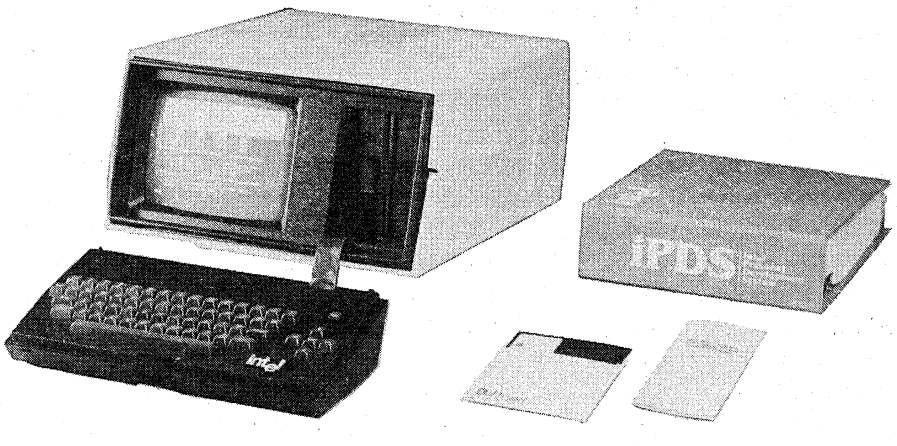
Documentation Supplied

- *Intellec Series IV Microcomputer Development System Overview,* Order Number 121752
- *Intellec Series IV Microcomputer Development System Installation and Checkout Manual,* Order Number 121757
- *Intellec Series IV Operating and Programming Guide,* Order Number 121753
- *Intellec Series IV Pocket Reference,* Order Number 121760
- *Intellec Series IVC ISIS-IV User's Guide,* Order Number 121880
- *Intellec Series IV ISIS-IV Pocket Reference,* Order Number 121890
- *AEDIT Text Editor User's Guide,* Order Number 121756
- *AEDIT Text Editor Pocket Reference,* Order Number 121767
- *DEBUG-88 User's Guide,* Order Number 121758
- *iAPX 88 Book,* Order Number 210200
- *iAPX 86, 88 User's Manual,* Order Number 210201
- *iAPX 86, 88 Family Utilities User's Guide,* Order Number 121616
- *MCS-80/85 Family User's Manual,* Order Number 121506
- *MCS-80/85 Utilities User's Guide for 8080/8085-Based Development Systems,* Order Number 121617
- *8080/8085 Floating-Point Arithmetic Library User's Manual,* Order Number 9800452
- *An Introduction to ASM86,* Order Number 121689
- *ASM86 Macro Assembler Operating Instructions for 8086-Based Systems,* Order Number 121628
- *ASM86 Language Reference Manual,* Order Number 121703
- *ASM86 Macro Assembler Pocket Reference,* Order Number 121674

iPDS™
PERSONAL DEVELOPMENT SYSTEM

- **Completely Integrated Computer System Packaged in a Compact Rugged Enclosure for Portability**
- **Comprehensive Design Tool for 8-Bit Microprocessors and Microcontrollers**
- **Microprocessor Emulator (EMV) Functions**
- **EPROM Programming Functions**
- **Dual Processing Capability**
- **Expandable using Standard MULTIMODULE™ Cards**
- **Desk Top Computer for CP/M* Based Applications**
- **640 KByte Integral Flexible Disk Drive; Expandable to 2.56 Million Bytes**
- **Powerful ISIS-PDS Disk Operating System with Relocating Macro-Assembler, and CRT-Based Editor**
- **Optional High Level Languages FORTRAN 80, PL/M 80, PL/M 88/86 and Basic**
- **Software Compatible with Previous Intellec® Systems**
- **Bubble Memory Option**

The iPDS Development System is a completely integrated computer system supporting the development of products incorporating Intel microcontrollers or 8-bit microprocessors. Used with its optional emulation vehicles (EMVs) and iUP PROM Programming Personality Modules, the iPDS system provides comprehensive support for integrated hardware and software development, product testing during manufacture, and customer support after the product is in the field. The unit is designed with portability in mind permitting the iPDS Development System to be conveniently transported around the laboratory and into the field. Extensive software is available thereby simplifying and speeding up product development. The software is designed to make the iPDS system easy to use for the novice as well as satisfying the needs of the experienced user. Used with the optional CP/M operating system, the iPDS system becomes a desk top computer that can execute CP/M compatible application programs.



220390–1

*Registered Trademark of Digital Research Inc.

October 1986
Order Number: 210390-003

FUNCTIONAL DESCRIPTION

Hardware Components

The iPDS case comprises two high impact, shock resistant, poly-carbonate plastic enclosures, that when fitted together, provide a compact and fully enclosed unit. The main enclosure houses a CRT, flexible disk drive, power supply, and base processor printed board assembly. The second enclosure houses the keyboard. On the right side of the unit a spring loaded door allows insertion of an emulator module or an iUP PROM programming module. On the top, a hinged panel covers the storage space for cables and plug-in modules. The carrying handle is attached to the front of the main enclosure and folds away when the system is in use. In the closed position, the iPDS system is 8.15" high, 16" wide, 20" long, and conveniently fits under an airline seat. The basic unit weighs 27 pounds.

BASE PROCESSOR PRINTED BOARD ASSEMBLY-BPB

The Base Processor Board (BPB) contains the powerful 8085A microprocessor, 64 Kbytes of RAM, CRT/keyboard controller, floppy disk controller, serial I/O port, and parallel I/O port. There are interfaces for connection to the Optional Processor Board, MULTIMODULE Adaptor Board, and the EMV/PROM Programming Adaptor Board.

INTEGRAL CRT

The CRT is a 9 inch green phosphor (P42) unit that displays 24 lines of 80 characters/line with a nominal 15.6 KHz vertical sweep rate. The CRT controller, based on an Intel 8085 and 8275 Programmable Controller Chip is located on the BPB. A single cable containing the signals, power, and ground connects it to the CRT. The contrast adjustment is accessible at the rear of the unit. A pull out bail allows the CRT to be placed in a comfortable operating position of 24 degrees to the horizontal. The standard ASCII set of 94 printable characters is displayable, including upper and lower case alpha characters, and the digits 0 through 9. Another 31 characters for character graphics are defined. If the Optional Processor Board is installed, the second processor shares the CRT with the base processor. The bottom part of the screen is assigned to the processor communicating with the keyboard. The top part of the screen displayed in reverse video is assigned to the other processor. The number of lines appearing on the screen for each processor can be completely controlled by the user via special function keys.

KEYBOARD

The keyboard is housed in a separate enclosure and a flat shielded cable connects it directly to the keyboard controller on the BPB. This 5" cable provides the flexibility to place the keyboard in a comfortable operating position relative to the main enclosure. A total of 61 keys include a typewriter keyset, cursor control keys, and function keys. Auto repeat is available for all keys and is implemented by the keyboard controller. If the Optional Processor Board is installed, it shares the keyboard with the base processor. Initially, the keyboard is assigned to the base processor. It can be assigned to the optional processor by pressing the special function key, FUNC-HOME. Subsequent use of the FUNC-HOME key alternates the keyboard assignment between the two processors.

INTEGRAL FLOPPY DISK DRIVE

The integral floppy disk drive is a 5¼", double-sided, 96 tracks-per-inch drive. Diskettes are written double-sided, double density and provide 640 Kbytes of formatted storage in the built-in drive. The floppy disk controller located on the BPB is based on the Intel 8272 floppy disk controller chip, and can control three additional drives. The ISIS-PDS operating system supports the disk drives. If the Optional Processor Board is installed, the integral disk drive is shared by the two processors or it can be exclusively assigned to one of the processors. When shared, only one processor can access a drive at a time. However, the disk drive sharing is transparent to the user since the ISIS-PDS operating system controls the accessing of the drive and automatically resolves file contention.

INPUT/OUTPUT

The iPDS Development System contains two I/O channels located at the rear of the base enclosure and wired to the I/O ports on the base processor board. The serial channel is an EIA RS-232-C interface for asynchronous and synchronous data transfer and is based on the Intel 8251 USART and 8253 timer. The interface can be software configured using the SERIAL command. Full duplex asynchronous operation from 110 to 19.2K baud is selectable.

The parallel I/O interface is an 8-bit parallel I/O port supporting a Centronics type printer. The interface is implemented with an Intel 8255 Programmable Parallel Interface chip. A maximum transfer rate of 600 cps is supported.

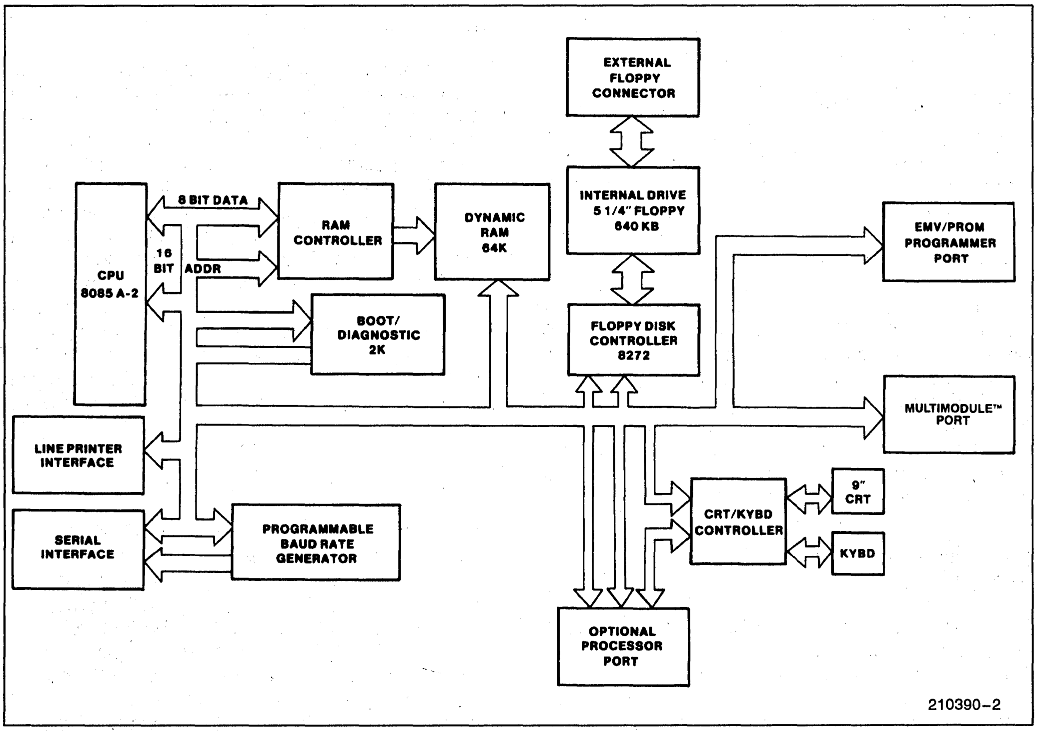


Figure 1. iPDS™ Block Diagram

Software Components

ISIS-PDS OPERATING SYSTEM

The ISIS-PDS operating system included with the basic iPDS system is designed with a major emphasis on ease of use and simplification of microcomputer development. It is based on the proven ISIS II operating system available on all Intellec Microcomputer Development Systems.

ISIS-PDS has a comprehensive set of commands to control system operation. These commands can be divided into five functional groups.

- System Management Commands
- Device Management Commands
- File Management Commands
- Program Development Commands
- Program Execution Commands

Table 1 summarizes these commands. The HELP commands are especially useful, providing the user with on-line assistance, eliminating frequent referencing of the manual.

ISIS-PDS CREDIT™ TEXT EDITOR

Included with iPDS is the Intel CRT-based text editor, CREDIT. It is used to create and edit ASCII text files on the Intel Personal Development System. Once the text has been edited, it can be directed to the appropriate language processor for compilation, assembly, or interpretation. CREDIT features, shown in Table 2, are easy to use and simplify the editing and manipulation of text files.

The two editing modes in CREDIT are screen mode and line mode. In screen mode and text being edited is displayed on the CRT and corrected by either typing the new text or using the single stroke character

	SYSTEM MANAGEMENT COMMANDS
HELP	displays help information for operating system commands.
?	displays the version number of the current Command Line Interpreter.
FUNC-R	software resets the processor to which the keyboard is currently assigned.
FUNC-S	switches the CRT display speed between a slow and fast speed.
FUNC-T	switches the keyboard between typewriter mode and locked upper case mode.
FUNC-HOME	switches the current foreground and background processors.
FUNC-↑	increases the display for the foreground processor by one line and decreases the background processor display by one line.
FUNC-↓	decreases the display for the foreground processor by one line and increases the background processor display by one line.
	DEVICE MANAGEMENT COMMANDS
IDISK	initially prepares disks and bubble memory for use with the operating system.
ASSIGN	displays or assigns the mapping of physical to logical devices.
#	re-assigns the system output to the CRT display screen.
FUNC<n>	changes the system input from the keyboard to the file named JOB <n> CSD where <n> is a one-digit number from 0 to 9.
/	changes the system input from the keyboard to a file or device which is specified by the user.
SERIAL	initializes the serial I/O port.
ATTACH	assigns a row of multimodules to a processor.
DETACH	releases a row of multimodules from a processor.
	FILE MANAGEMENT COMMANDS
DIR	displays a list of the files stored on a disk or on bubble memory.
ATTRIB	displays and modifies the attributes of a file.
COPY	transfers files and appends files.
DELETE	removes files from the disk.
RENAME	changes the filename and/or extension of a file.
@	displays the contents of a file on the screen.
	PROGRAM DEVELOPMENT COMMANDS
LIB	allows the user to manage a library of MSC-80/85 program modules.
LINK	combines a number of object modules into a single object module in an output file
LOCATE	converts relocatable object programs into absolute object programs by supplying memory addresses throughout the program.
HEXOBJ	converts a program from hexadecimal file format to absolute object format.
OBJHEX	converts a program from absolute object format to hexadecimal file format.
DEBUG	provides a minimum set of 8080/8085 debugging commands.

Table 1. Functional Summary of ISIS-PDS Commands

PROGRAM EXECUTION COMMANDS	
<filename>	loads and executes the object program named <filename>.
SUBMIT	reads an input SUBMIT file, creates a command file containing ISIS commands, and executes commands in sequence from the file created.
•	is a fast form of the SUBMIT command. One command line is read from the SUBMIT file, transformed into an ISIS command in memory, and executed. No intermediate file is created.
/	reads ISIS commands from a disk job file and executes them in sequence. The / command is also considered a device management command.
JOB	stores a sequence of frequently used ISIS commands in a job file as they are entered from the keyboard without executing them until the sequence is completely entered. Two job files, ABOOT.CSD and BBOOT.CSD, deserve special mention. If either of these files is present (ABOOT.CSD for Processor A and BBOOT.CSD for Processor B) when the system is initialized, commands are automatically executed from the file. This feature can be used to configure a system.
ENDJOB	stops the automatic execution of commands from a JOB file and returns control to the keyboard.
ESC	edits the previously entered or the current command line and allows the new command line to be executed.

Table 1. Functional Summary of ISIS-PDS Commands (Continued)

control keys. Single character control keys are used for changing, deleting, inserting, paging forward, and paging backwards.

In command line mode, high level commands are used for complex editing. Examples of the functions available in the command line mode are searching, block moves, copying, macro definitions, and manipulating external files.

The AEDIT text editor used in all Intellec Development Systems, is available as an option for the iPDS System.

8080/85 MACRO ASSEMBLER

The iPDS also includes the INTEL 8080/85 Macro Assembler. This marco assembler translates programs written in 8080/8085 assembly language to the machine language of the microprocessor. It also produces debug data. The debug utility can be used

CREDIT™ Editor features two editing modes: cursor-driven screen editing and command line context editing

CRT Editing Includes:

- Displays full page of text
- Single control key commands for insertion, deletion, page forward and backward
- Type-over correction and replacement
- Immediate feedback of the results of each operation
- The current state of the text is always represented on the display

Command Line Editing Includes:

- String search and substitute
- String delete, change, or insert
- Block move
- Block copy
- User-defined macros
- External file handling
- Change CREDIT features with ALTER command
- Conditional iteration
- Use-defined tab settings
- Symbolic tag positions
- Automatic disk full warning
- Runs under ISIS-II SUBMIT facility
- Option to exit at any time with original file intact
- HELP command

Table 2. Summary of CREDIT™ Editor Features

to troubleshoot the assembler-produced machine language using features such as software breakpoints, single step execution, register display, disassembly, and I/O port access. This reduces the time spent troubleshooting the software and supports modular program development.

UTILITIES

Utility programs included with iPDS are: DEBUG, LIBRARY, LINK and LOCATE. These programs aid in software development and make it possible to combine programs and prepare them for execution from any memory location.

DIAGNOSTICS

The iPDS System includes system diagnostic routines executed during system initialization. These routines verify the correct operation of the system and aid the user in fault isolation. Any failures in the basic system components, base processor, CRT/Keyboard, optional processor, or the power supply are indicated by four diagnostic LED indicators mounted on the base processor boards. These LED's are viewed through the spring loaded door on the right side of the unit. When basic system components are operational, additional errors are indicated by messages to the CRT display screen.

After ISIS-PDS is loaded and started, additional confidence tests are available to verify correct system operation. These tests included on the system disk, run as utilities under the operating system and can be selectively executed to verify individual functions on the main processor board, optional processor board, bubble memory MULTIMODULEs and EMV/PROM programmer adaptor.

iPDS™ HARDWARE OPTIONS

Add-On Mass Storage

Mass storage can be increased by adding up to three external flexible disk drives. This adds 640

Kbytes of formatted mass storage per additional drive. The maximum disk storage available on iPDS is 2.56 Mbytes. The optional drive is vertically mounted and housed in a plastic enclosure with its own power supply. A 20" cable connects the optional floppy drives to the external disk drive connector on the rear of the iPDS system.

The iPDS system also supports Intel's iSBX™ 251 Bubble Memory MULTIMODULE. A maximum of two bubble MULTIMODULEs can be added. Each contain 128 Kbytes of non-volatile memory. Bubble emory MULTIMODULEs can only be added to a system containing the MULTIMODULE Adaptor Board. The bubble memory is treated by the ISIS-PDS and CP/M operating system as an additional disk drive with the same file structure and directory structure as a diskette. The bootstrap ROM is programmed to boot the operating system from the bubble. The iSBX 251 MULTIMODULE has no moving parts, making it ideal for applications where ruggedness is an important consideration. The bubble memory is also recommended for systems requiring portability, since it is completely enclosed in the iPDS main unit.

Optional Processor Board

The Optional Processor Board provides dual processing capabilities and increases the processor power of the iPDS system. A different program can be run on each of the processors at the same time, providing a greater processing throughput. Each processor operates under ISIS-PDS control. The Optional Processor Board also provides a convenience feature for accessing directories, file displays and HELP without interrupting the main processor task.

The Optional Processor Board contains functions identical to the base processor. There is an 8085A CPU with 64 Kbytes of dynamic RAM and an additional 2 Kbytes of bootstrap ROM.

Both processors share the keyboard, the CRT display unit, the disk drives, and the MULTIMODULEs. Serial or parallel I/O ports can be added to the optional processor through iSBX MULTIMODULEs. Each processor runs the ISIS-PDS operating system and applications programs in its own 64 Kbyte memory space, independent of the other processor. Special hardware function keys are provided to facilitate procedures necessary in the dual processing environment. These procedures include independent initialization of each processor, sharing of the CRT display, and assignment of the keyboard. The ISIS-PDS commands facilitate sharing of disk drives, MULTIMODULEs, and files.

Emulation Vehicles (EMVs)

Emulation vehicles (EMVs) for use with the iPDS Development System, are available for debugging a variety of Intel microprocessor families, such as the 8088, 8051, or 8044. Emulators consist of hardware and software. The EMV hardware is inserted into the EMV/iUP Personality Module port of the iPDS System. The optional EMV/Prom Programming Adaptor Board is required to install the EMV's. The emulator software runs under the ISIS-PDS operating system and provides the user's interface to the emulator.

An EMV contains features used to debug microprocessor designs quickly and efficiently. It provides a controlled environment for exercising a user design and monitoring the results. It exactly duplicates the behavior of a target microprocessor/microcontoller in the user's prototype system while providing information to the user to aid in integrated hardware and software development. EMV's provide features for real time full speed emulation as well as single step execution of a user's design. Breakpoint features allow the user to specify a portion of the program to execute and then stop for interrogation. During execution, the EMV automatically collects execution history in the trace buffer. Once stopped at the breakpoint, the emulator acts as a window to the internal registers and logic signals inaccessible from the connector pins. This provides for examination and alteration of the internal state of the microprocessor.

The emulator accepts symbolic debug data, such as symbol tables produced by the language translators. Therefore, when debugging, the programmer can reference locations in the program elements with the symbol names used in the source program, rather than absolute memory addresses.

Another advantage of using an emulator is functional prototype hardware is not required to begin software debugging. The emulator duplicates the behavior of the target microprocessor and provides some resources, such as memory, that can be used until the hardware prototype is closer to completion.

The software controlling the emulator comprises a set of commands the user enters to directly control interactive debugging sessions. The command families are listed in Table 3. Also, sequences of emulator commands can be executed automatically from a file, providing a basis for manufacturing and field test routines.

Emulation Commands

BR-Display breakpoint menu
BR0, 1, 2, 3-Change/display breakpoint register for execution address
BRR-Change/display breakpoint register for execution range
BRB-Change/display break on branch
BV-Change/display break on value
BC-Clear all breaks
TB0, 1, 2, 3-Enable/disable display by bit value
TR0, 1, 2, 3-Enable/disable display by execution address
TV-Enable/disable display by register value
TR-Enable/disable display of registers
TS-Enable/disable display of PSW
TD-Enable/disable display of code disassembly
STEP-Enter slow down emulation mode
GO-Enter real-time emulation mode

Advanced Commands

MACRO-define, and display macro
IF THEN
COUNT
REPEAT } CONTROL CONSTRUCTS
WHILE
UNTIL
FUNCTION KEY-invoke macro assigned to function key

Utility Commands

HELP-Displays command syntax
LOAD-Loads object file in mapped memory
LIST-Generates copy of emulation work session
DEFINE-Defines symbol or macro
SYMBOL-Displays symbols
REMOVE-Deletes symbol or macro
ENABLE/DISABLE-Control for expanded display
EVALUATE-Evaluate any expression
SUFFIX/BASE-Sets input and display numeric base
SAVE-Save code memory to file
RESET-Resets emulation processor
EXIT-Terminate emulation session

Display/Modify Commands

REGISTER-Menu for change/display registers
MEMORY-Menu for change/display memory
DUMP-Display memory as ASCII and Hexadecimal
ASM/DASM-change/display code memory as assembly language mnemonics

Table 3. Summary of Typical Emulator Commands

Plug-in emulators with identical user characteristics and software to Intel's EMV products are also available from third party vendors for additional microprocessors, such as the 8085.

iUP Personality Modules

The iPDS System accepts most Intel PROM Programming Personality Modules from our iUP-200A/201A product line. These modules provide all the hardware and firmware needed for programming entire families of Intel EPROMS, E^{2}PROMS, and microcontrollers containing on-chip EPROM. The optional EMV/PROM Programming Adaptor Board is required to use the iUP Personality Modules. Intel Prom Programming Software (IPPS) runs under the ISIS-PDS operating system and is included with the EMV/PROM Programming Adaptor Module. This software provides a set of commands to control the programming and verification of the devices.

EMV/PROM Programming Adaptor Board

The EMV/PROM Programming Adaptor Board provides an interface between the Base Processor Board and EMV or PROM programming modules. This option is required before either of these modules can be operated with the iPDS.

MULTIMODULE™ BOARDS

The iPDS is expanded by utilizing a variety of Intel iSBX MULTIMODULE boards. The MULTIMODULE Adaptor Board allows a maximum of four MULTIMODULE boards to be added. MULTIMODULE boards are small, special function boards using the iSBX bus to interface to the CPU. The available iSBX MULTIMODULE boards include:

- iSBX 251 Bubble Memory MULTIMODULE Board
- iSBX 350 Parallel Port MULTIMODULE Board
- iSBX 351 Serial Port MULTIMODULE Board
- iSBX 488 IEEE-488 Interface MULTIMODULE Board
- iSBX 344 BITBUS™ Controller MULTIMODULE Board

The Insite™ Software Library contains many software routines for these MULTIMODULEs. The iPDS user manual contains technical information for writing custom I/O driver routines.

MULTIMODULE™ Adapter Board

The MULTIMODULE Adapter Board provides an interface between the base processor board and the MULTIMODULE options. It is required before any MULTIMODULE options can operate with the iPDS system.

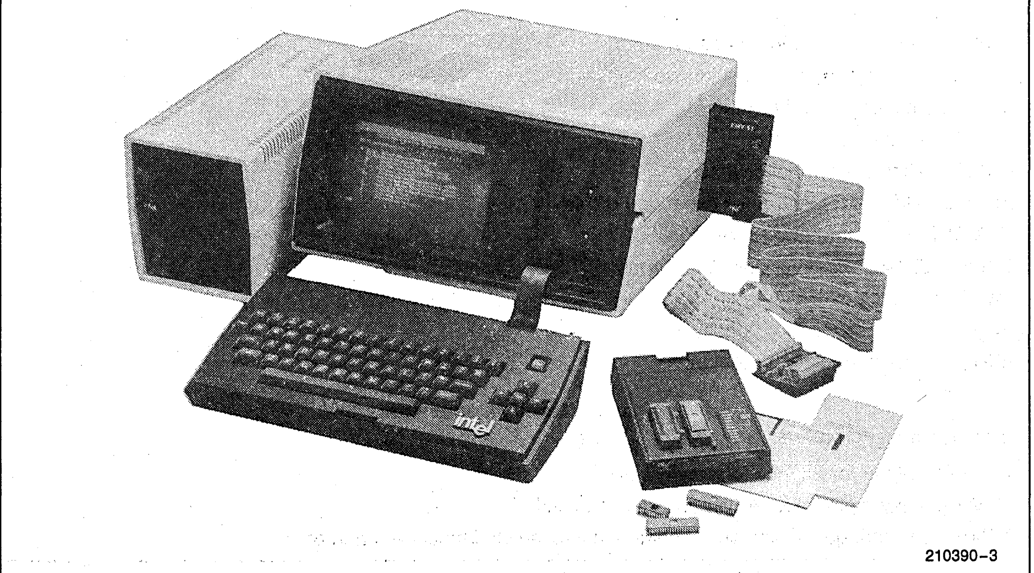


Figure 2. iPDS™ System with Optional Modules Installed

iPDS™ SOFTWARE OPTIONS

High Level Languages

High level languages help reduce system design effort and maintenance cost by allowing the programmer to design software at a more abstract level. A block structured language. PL/M 80, is available for the 8085, along with FORTRAN 80, PASCAL 80 and BASIC 80.

Software Support for Additional Microprocessor

Assemblers and high level languages for different target microprocessors are available to aid the software development effort. These include ASM-51, PL/M 88/86, ASM 88/86, ASM 8048/49, and PL/M 51.

General Purpose Computing Software

The iPDS can also be used as a general purpose desk top computer. The widely used CP/M microcomputer operating system is available for the iPDS from Intel. It supports iPDS systems with single or multiple disk drives, and iPDS systems using bubble memory for mass storage. CP/M compatible software will come from three sources; vendors of CP/M based software programs, independent software makers, and Intel. The software programs available from Intel include high level languages, wordprocessing software and an electronic spreadsheet.

File Transfer Package

Transferring files between the iPDS system and any of Intel's Intellec Development System is accomplished using the iPDS-FTRANS option. This product uploads/downloads files via the RS232C serial link and under control of software running on both the iPDS and the Intellec system. Data transmission is monitored and any errors are displayed. Transfer rates up to 19.2K baud can be selected. FTRANS can also be used to transfer files between remote systems using telephone modems.

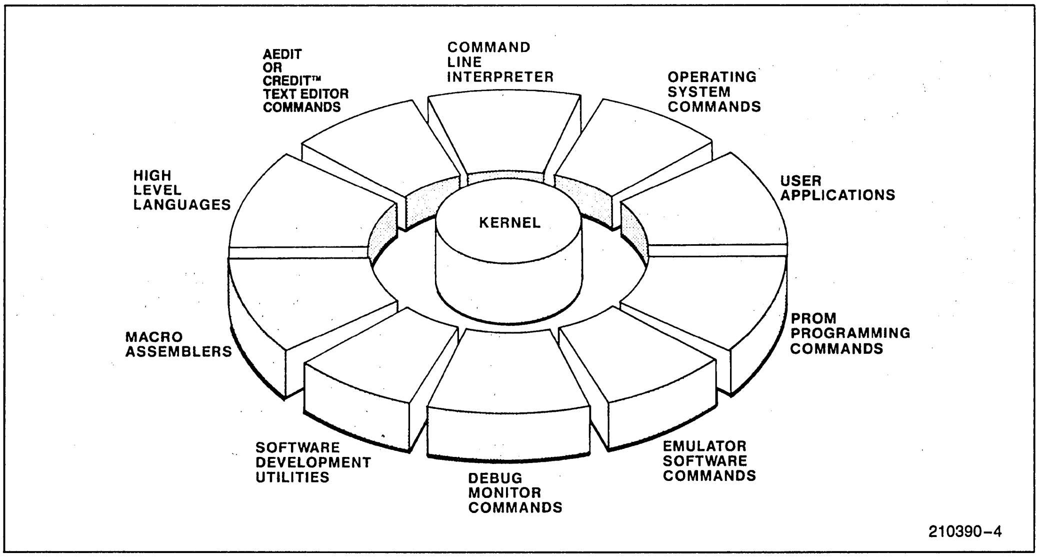

AEDIT OR CREDIT™ TEXT EDITOR COMMANDS
COMMAND LINE INTERPRETER
OPERATING SYSTEM COMMANDS
HIGH LEVEL LANGUAGES
USER APPLICATIONS
KERNEL
MACRO ASSEMBLERS
PROM PROGRAMMING COMMANDS
SOFTWARE DEVELOPMENT UTILITIES
DEBUG MONITOR COMMANDS
EMULATOR SOFTWARE COMMANDS
210390-4

Figure 3. Overview of iPDS™ System Software Environment

SPECIFICATIONS

Host Processor

8085A-2 based, operating at 5.0 MHz

Memory

RAM-64K of user memory on BPB
ROM-2K (Boot/diagnostic)

I/O Interfaces

I/O Serial Channel; RS-232 at 110–19.2K baud (asynchronous) or 150-56K baud (synchronous). Baud rate and serial format software controllable.

I/O Parallel Channel; 8-bit parallel supporting Centronics type printer. Transfer rate up to 600 characters per second.

Memory Access Time

RAM-450 ns.

Integral Flexible Disk Drive

System Storage Capacity
DS/DD-640 Kbytes (formatted)

Data Transfer Rate
250 Kbits/sec.

System Access Time
Track to Track: 6 msec.
Rotational Speed: 300 rpm
Motor Start Time: 0.4 sec. max

Media
5¼″ disk with 1 index hole

Physical Characteristics

Closed Unit (without options)

Height:	8.15 in.
Width:	16 in.
Depth:	20 in.
Weight:	27 lbs.

Power Requirement

Input Voltage:
115/220 VAC Selectable Single Phase
115 VAC (90 VAC-132 VAC) 47–63 Hz, 1 amp
220 VAC (180 VAC-264 VAC) 47–63 Hz, 0.5 amp

Optional Electrical Requirements

Optional Electrical Requirements (Max. In Amperes)										
Power Supply Voltage	Optional Processor	EMV/PROM Adaptor	MULTIMODULE™ Adaptor	ISBX™ 350 BOARD	ISBX™ 351 BOARD	ISBX™ 251 BOARD	ISBX™ 488 BOARD	EMVs	IUP
+5 volts	1.0	0.3	0.6	0.62	0.53	0.37	0.6	2.5	0.7
+12 volts	—	0.18	—	—	0.03	0.4	—	—	0.85
−12 volts	—	0.05	—	—	0.03	—	—	—	0.4

Maximum option power requirements must not exceed 33.6 watts for any configuration.

ENVIRONMENTAL CHARACTERISTICS

Operating

Temperature:	10°C to 30°C
Relative Humidity:	20% to 80%
Maximum wet bulb:	25.6°C

Non-Operating

Temperature:	40°C to 62°C
Relative Humidity:	5% to 95% (non-condensing)

Operating Vibration

0 to 0.004 inches peak to peak excursion from 10 to 55 Hz.

Non-Operating Shock

15 G with shock wave of 20 ms duration, ½ sine wave.

Equipment Supplied

iPDS System Enclosure including:

- Base Processor Board (BPB)
- CRT/Keyboard
- Integral Flopy Disk Drive
- System Diskette with ISIS-PDS operating system
- MSC-80/MSC-85 Macro Assembler
- Debug-85, Link, Locate and Library Utilities
- CREDIT CRT-based text editor
- System confidence tests

iPDS System Literature Kit including:

- Intel Personal Development System User's Guide 162606
- Intel Personal Development System Pocket Reference 162607
- 8080/8085 Assembly Language Programming Manual 9800301
- 8080/8085 Assembly Language Reference Card 9800438
- MSC-8085 Utilities User's Guide for 8080/8085 Based Development System 121671
- ISIS II 8080/8085 Macro Assembly Operating Manual 9800292

Reference Manuals

- ISIS-II System User's Guide 9800306
- iPDS Demonstration Kit 210745-002

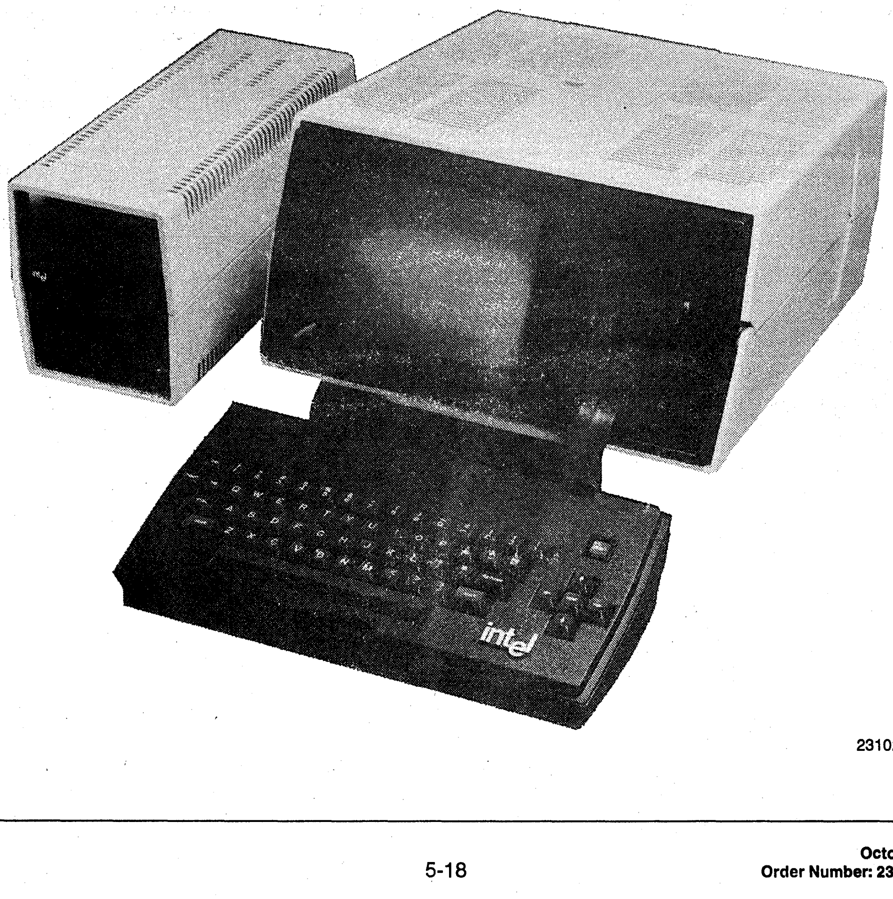
ORDERING INFORMATION

Part Number	Description
iPDS-100	iPDS System
iPDS-110	Optional Processor Board
iPDS-120	MULTIMODULE Adapter Board
iPDS-130	Add-On Disk Drive
iPDS-140	EMV/PROM Programming Adaptor Board
iPDS-FTRAMS	iPDS/iMDX File Transfer Package
iPDS-PROTO KIT	Design aid for developing plug-ins

THE iPDS™-130 OPTIONAL FLEXIBLE EXTERNAL DISK DRIVE FOR THE iPDS PERSONAL DEVELOPMENT SYSTEM

- **Each Disk Drive Provides 640 Kbytes of Formatted Mass Storage**
- **Daisy-Chaining up to 3 Disk Drives Provides a Total of 2.56 Mbytes Storage Capacity**
- **Each Disk Drive Has Its Own Power Supply**
- **Disk Drives are Industry-Standard 5¼ Inch Flexible Diskettes as the Storage Medium**
- **Disk Drive Has Transfer Rate of 4 μs/Bit, a Recording Density of 5922 bpi, and Dual Heads**
- **Use of External Disk Drive Eliminates Disk Swapping when Making Duplicate Disks**

When using the iPDS personal development system, applications may be developed that require more storage capacity than is provided by the integral disk drive of the system. The iPDS-130 optional external flexible disk drive provides the needed additional mass storage. Up to three disk drives may be added to the iPDS system, with each additional disk drive providing 640 Kbytes of (formatted) capacity. This means that a maximum disk storage of 2.56 Mbytes is available. The photograph below shows the iPDS-130 external disk drive with the iPDS system. Figure 1 shows some features of the iPDS-130 disk Drive.


231020–1

October 1986
Order Number: 231020-001

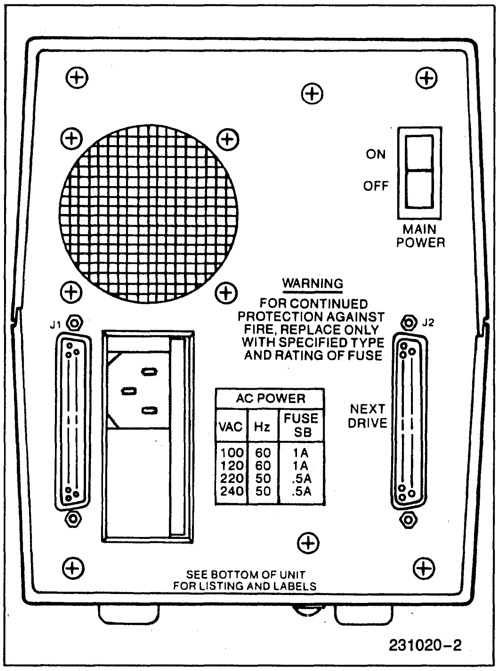

ON
OFF
MAIN POWER
WARNING
FOR CONTINUED PROTECTION AGAINST FIRE, REPLACE ONLY WITH SPECIFIED TYPE AND RATING OF FUSE
J1
J2
AC POWER
VAC
Hz
FUSE SB
100 60 1A
120 60 1A
220 50 .5A
240 50 .5A
NEXT DRIVE
SEE BOTTOM OF UNIT FOR LISTING AND LABELS
231020-2

Figure 3. iPDS™-130 Optional Flexible External Disk Drive Rear Panel

Power Supply

The flexible disk drive unit contains a linear power supply with a maximum power input of 40 watts. The output consists of two regulated dc voltages (5V and 12V).

I/O SPECIFICATIONS

Floppy Disk Interface

The floppy disk interface controls up to four 5-¼ in. double-sided 96 tpi floppy disk drives.

The floppy disk is a 5-¼ in., 96 tpi, dual-headed unit. With a total of 80 tracks of sixteen 256-byte sectors per side, the formatted capacity of the unit is 640 Kbytes. The interface is the industry standard for 5-¼ in. drives.

OPTIONAL FLEXIBLE EXTERNAL DISK DRIVE SPECIFICATIONS

The specifications for the optional flexible external disk drive are given in Tables 1 through 4.

Table 1. Environmental Characteristics

Temperature	
Operating	10°C to 35°C
Non-operating	−40°C to 62°C
Humidity	
Operating	20% to 80% non-condensing
Non-operating	5% to 95% non-condensing
Cooling	Up to 60 watts are dissipated by fan cooling

Table 2. Physical Characteristics

Width	6.1 in (155.4 mm)
Height	7.3 in (174.2 mm)
Depth	13.8 in. (350.6 mm)
Weight	11.0 lbs. (5.0 kg)

Table 3. Electrical Characteristics

Input Power	90 VAC to 132 VAC, 47 Hz to 63 Hz; or 198 VAC to 264 VAC, 47 Hz to 63 Hz
Drive Power	12 VDC ±1%
Logic Power	5 VDC ±1%
Adjustable Range	±5%, drive and logic
Power Dissipation	25 watts average, 34 watts maximum

Table 4. Functional Specifications

Transfer Rate	4μs/bit
Rotational Speed	300 rpm ±1.5%
Track Density	96 tpi
Number of Cylinders	80
Number of Sides	2
Recording Density	5922 bpi
Encoding Method	MFM
Unformatted Capacity	6.25 Kbytes/track
Formatted Capacity	640 Kbytes
Motor Start Time	0.4 sec Maximum
Track-to-Track Step Rate	6 ms Maximum
Side-to-Side Delay Time	0.2 ms Maximum
Head Loading Time	35 ms Maximum
Head Setting Time	15 ms Maximum
Medium	Industry standard 5-¼ in. with Single Hole

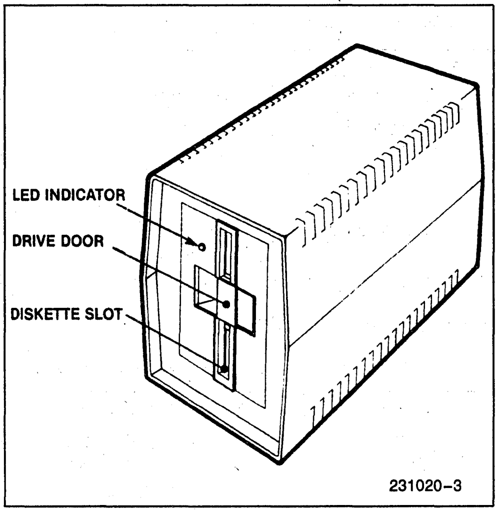


Figure 1. iPDS™-130 Flexible Disk Drive

Creating back-up diskettes is good programming practice and the iPDS-130 disk drive provides the means to create these back-ups. It shortens the time required and lessens the trouble associated with this task by eliminating the need to swap disks during the duplication process. The master diskette can be inserted in the iPDS system's integral disk drive and the duplicate diskette in the external disk drive.

The first external disk drive attaches to the rear of the main enclosure, and the other two external drives are connected to the rear of the previous external drive. Each additional drive has its own power supply and is mounted in its own housing. Figure 2 shows the iPDS unit with all three external drives.

HARDWARE

Each drive is 7.3 in. high and weighs approximately 11 lbs. The front of each disk drive contains a door, a door release mechanism, and a drive indicator that is lit during disk I/O operations. The drive is mounted in the vertical position. Different ac voltage ranges may be selected. The rear panel of the drive contains the ac power connector, the power ON/OFF switch, a fuse holder, a voltage selector card, and two I/O cable connectors. Figure 3 shows the disk drive's rear panel.

I/O Cable

The I/O cable is used to interconnect the iPDS system and the external disk drives. The external portion of the input cable is 30 in. long and connects to the flexible disk connector on the rear of either the iPDS unit or the previous optional disk drive. The output connector of the daisy-chain mounts on the rear panel of the disk drive and provides the connector to the next disk drive.

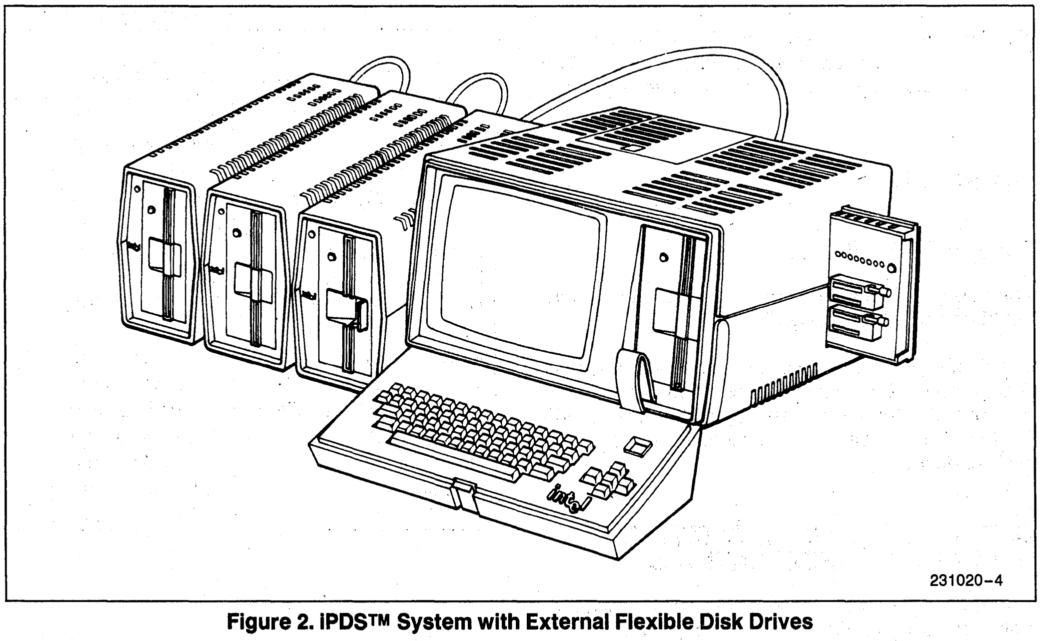


Figure 2. iPDS™ System with External Flexible Disk Drives

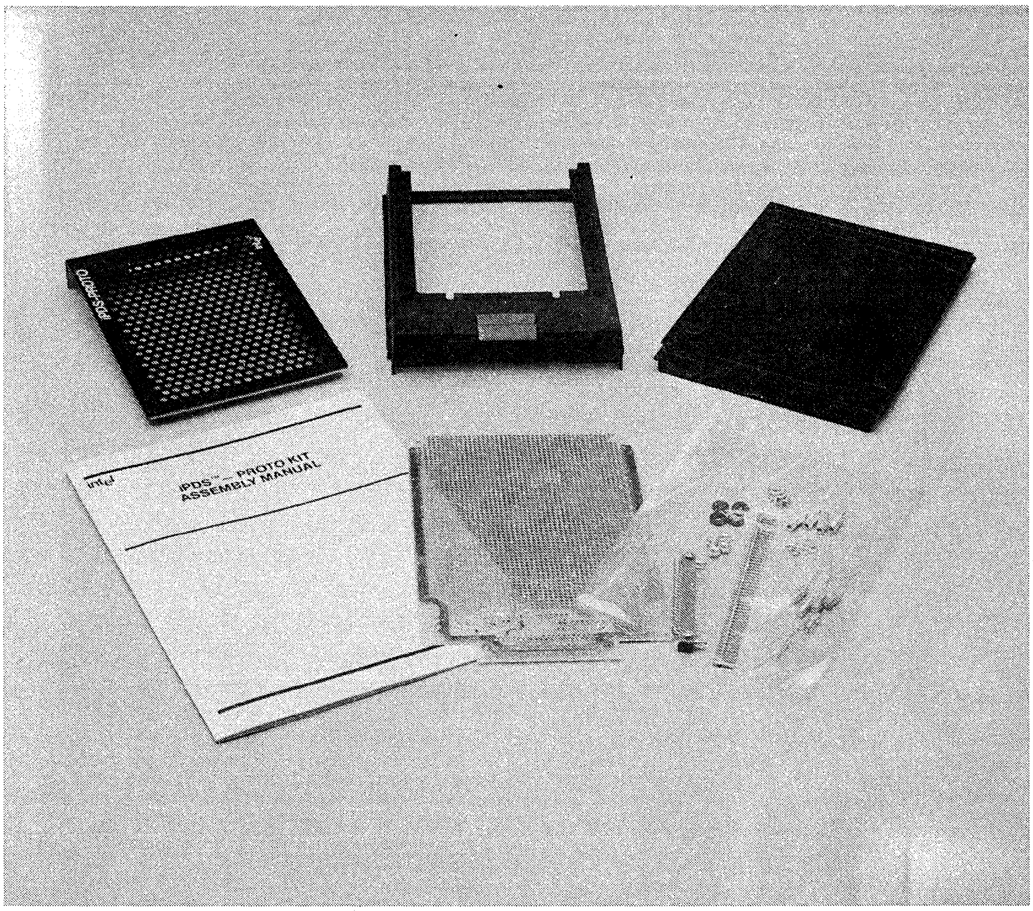
ORDERING INFORMATION

Part Number	Description
iPDS-130	Optional external flexible disk drive

iPDS™-PROTO KIT

- **Design Aid for Developing Your Own Specialized Plug-In Modules for the iPDS™ Development System and for the iUP-200/201 System, such as:**
  - **— Emulation Vehicle (EMV) Modules**
  - **— PROM or Programmed Logic Array (PLA) Programming Modules**
  - **— Instrumentation Modules (Logic or Signature Analyzers)**
  - **— Specialized Communications Modules**
  - **— Analog Interface Modules**
  - **— Program Storage Modules**
- **iPDX Bus Interface**
- **Easy-to-Follow Assembly Instructions**

The iPDS-PROTO Kit is a complete kit for engineers who want to enhance the iPDS development system and the iUP-200/201 Universal Programmer system by developing their own specialized plug-in modules such as those noted above. The module case and PROTO board are specifically designed to plug into both the iPDS system and the iUP-200/201 system.



280046–1

June 1986
Order Number: 280046-001

KIT COMPONENTS

The iPDS-PROTO Kit comprises the module case, the PROTO board, and a hardware kit. The hardware kit includes one iPDS bus connector, five isolation capacitors, wire-wrap pins, screws, washers, and lock nuts. Also included are the *iPDS™-PROTO Kit Assembly Manual* and the application note, *Designing Modules for the iPDS™ and iUP Systems* (Order Number #230682).

The PROTO board can accept up to 30 integrated circuits and associated discrete components.

iPDX BUS INTERFACE

The iPDX bus is a byte-wide, parallel interface between the plug-in module and the iPDS development system or the iUP-200/201 system. For further information on the iPDX bus, refer to the *Designing Modules for the iPDS™ and iUP Systems.*

ORDERING INFORMATION

Part Number	Description
iPDS-PROTO Kit	iPDS-PROTO board, module cover, hardware kit, and assembly manual.

intel®

APPLICATION NOTE

AP-156

October 1986

Designing Modules for iPDS™ and iUP Systems

DALE OLLILA
DSHO TECHNICAL PUBLICATIONS

Order Number: 230682-001

INTRODUCTION

The Intel Personal Development System (iPDS™) is a new development tool concept. It provides a subset of the capability of an Intellec® Series II/III development system, in a portable, and less expensive package. One of the features offered by the iPDS system is the expansion capability designed into the product. The basic iPDS system can be expanded to include a parallel processor, a wide range of serial (RS232C interface) and parallel (Centronics interface) devices, numerous MULTIMODULE™ (iSBX™ interface) devices, additional flexible disk drives, and a growing line of plug-in emulator and PROM programming modules.

The plug-in modules for the iPDS system communicate over an interface referred to as the Intel Personal Development Expansion bus (iPDS bus). The iPDX bus is also used in another Intel product, the iUP-200/201 Universal Programmer (iUP). There are some differences in iPDX bus implementation between the iUP and iPDS systems, but the basic interface is the same. Intel PROM programming modules can be used in either system.

THE iPDX BUS

The iPDX bus is a byte-wide, parallel interface between a plug-in module and the iPDS or the iUP system. The iPDX bus allows a variety of plug-in modules to be added to the iPDS system. (The iUP system normally is used with PROM programming modules.) Some of the possible types of plug-in modules are:

- PROM programming modules
- Emulator (EMV) modules for various microprocessor or microcontroller families
- Test instrumentation modules (e.g., logic or signature analyzers)
- Analog interface modules (e.g., analog/digital or digital/analog converters)
- Serial communication modules (e.g., modem or cassette controller modules)
- Parallel communication modules (e.g., direct interface to other CPU buses)
- Program storage modules (e.g., modules storing alternate operating systems, diagnostic programs, or games)

Intel Corporation produces plug-in modules that allow PROM programming and emulation for a variety of Intel chips. The special needs of individual users may not be satisfied by the plug-in modules that are available. This application note presents the specifications and design criteria for user-designed plug-in modules using the iPDX bus. User-designed plug-in modules can expand the usefulness of the iPDS system in the design lab, on the production floor, and in field applications.

iPDX Bus Features

The iPDX bus's capabilities are nearly equal to the capabilities of the iSBX™ bus. In some respects the iPDX bus is more powerful than the iSBX bus, due to the variable and switched supply voltages included on the bus. The features of the iPDX bus are:

- The controlling (iPDS or iUP) system supplies +5VDC and ground to the iPDX bus.
- The controlling (iPDS or iUP) system supplies switched voltages of +5.7VDC, −12VDC, and +8VDC to +27VDC to the plug-in modules. In addition, the iUP system controls a variable switched voltage (+8VDC to +15VDC) and the iPDS system controls a +12VDC switched voltage to the plug-in modules. The switched voltages are turned on and off under program control.
- A number of options are available for controlling iPDX bus transactions. These options include:

  1) Using iPPS software to supervise the uploading and execution of firmware from the plug-in module.

  2) Using a user-written driver program to supervise the uploading and execution of firmware from the plug-in module.

  3) Using a user-written driver program to control all iPDX bus activity.

  4) Using a user-written monitor program to allow control of iPDX bus activity from the system console.
- The plug-in modules that interface with the iPDX bus enable easy and fast changes of entire I/O subsystems.
- A prototyping tool (product code iPDS-PROTO) allows users to quickly design and build custom plug-in modules.
- The resources of a powerful, general-purpose development system (the iPDS system) are available to plug-in modules that use the iPDX bus.

Advantages and Limitations of iPDX Bus Implementation

The system (iUP or iPDS) that the iPDX bus is implemented on offers advantages for and imposes limitations on plug-in module use. The user's design requirements may dictate that the plug-in module be used with only one of the available systems. Plug-in modules that are universal must be designed to avoid the limitations of both systems.

iUP/iPDX BUS ADVANTAGES AND LIMITATIONS

Plug-in modules used with the iUP system are normally restricted to PROM-type programming functions. Table 1 lists the advantages and limitations of the iUP/iPDX Bus.

iPDS™/iPDX BUS ADVANTAGES AND LIMITATIONS

Plug-in modules used with the iPDS system can make use of all the features listed in the iPDX Bus Features section on page 4. The limitations for an iPDS/iPDX bus plug-in module are in the amount of power available from some of the voltage supply lines. Table 2 lists the advantages and limitations of the iPDS/iPDX Bus.

iPDX Bus Functional Description

The iPDX bus is an extension to the CPU bus of the iUP or iPDS system. The iPDX bus is active in the I/O address range 10H–1FH of the controlling CPU. Figure 1 is a functional block diagram of the iPDX bus as implemented on the iUP system. Figure 2 is a functional block diagram of the iPDX bus as implemented on the iPDS system.

iUP/iPDX BUS IMPLEMENTATION

The iPDX bus is the only I/O interface for the iUP-200/201 Universal Programmer, other than the serial interface of the iUP system. The iUP system normally performs one function, the programming of PROM-type devices. Intel PROM-type devices include EPROMs, E^{2}PROMs, and the EPROM portion

Table 1. iUP/iPDX Bus Implementations

Advantages	Limitations
The iUP system provides ample power for programming any type of PROM device.	Direct control of CPU operation is only possible using uploaded plug-in module firmware.
Two variable supply voltages are available for plug-in module use.	The V_{CC} line supplies a maximum of 1.0A to the plug-in module.
The I/O space of the iUP system is mostly unused, so operation in unused I/O space is possible.	

Table 2. iPDS™/iPDX Bus Implementations

Advantages	Limitations
The resources of the iPDS system (RAM, console, mass storage, etc.) are available to the plug-in.	Only one of the variable supply voltages (+VHSW) is available on the iPDS bus. The other variable line (+VLSW) has as fixed output of +12VDC.
The user has the option of using iPPS software or user-written programs to control the plug-in module.	Power supplied to the iPDX bus is not adequate for gang programming modules.
Any PROM programming module that works with the iPDS system and iPPS software also works with the iUP system. The V_{CC} supply line can handle up to 2.5A draw. This draw is adequate for most user applications.	

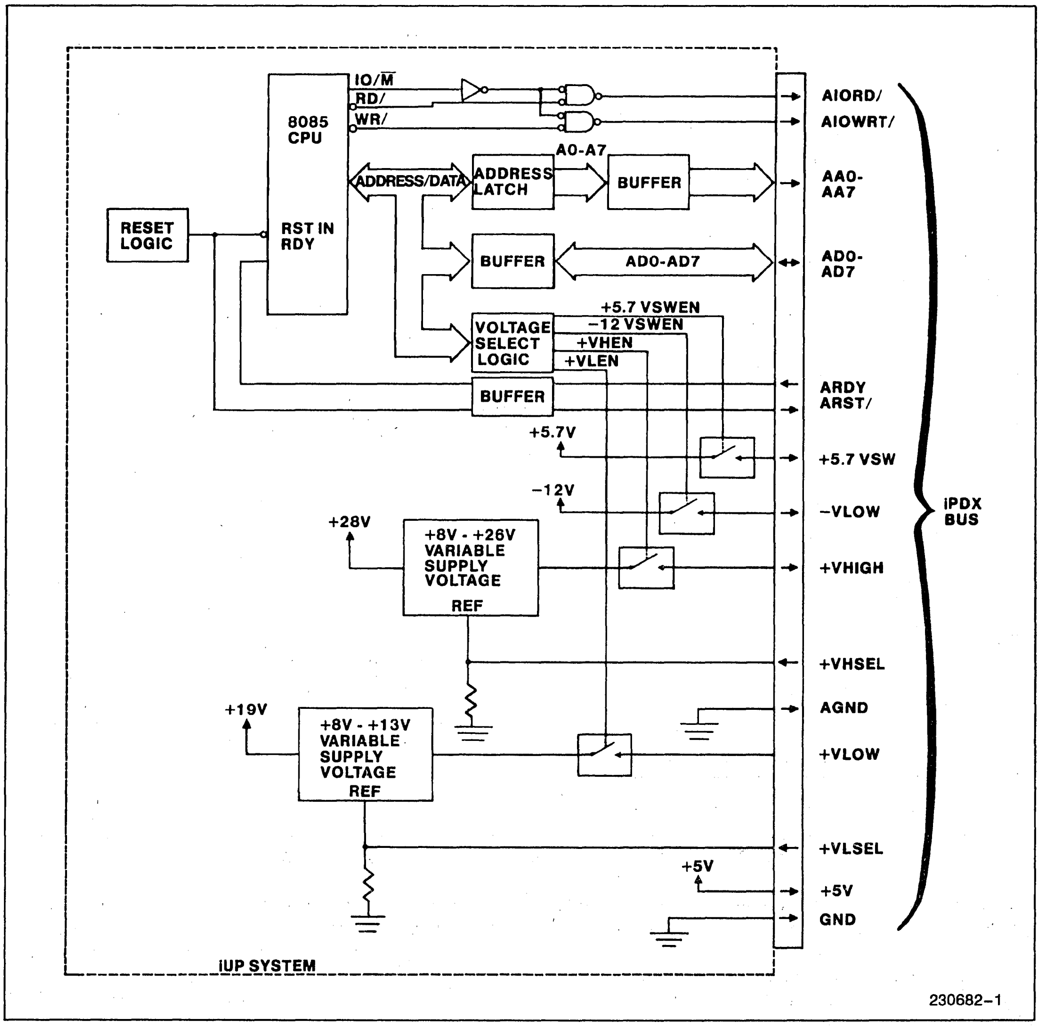


Figure 1. iUP/iPDX Bus, Functional Block Diagram

of various microcontrollers. The iUP system can program non-Intel PROM-type devices, but in most cases a personality plug-in module for the non-Intel device must be designed by the user. Note, however, that the Intel iUP-Fast 27/K PROM programming module (with firmware change) can program any 28-pin JEDEC device.

The iPDX bus implementation on the iUP system is optimized for maximum programming power capabilities. Each of the switched voltage supply lines from the iUP system provides at least twice the power of the corresponding line from an iPDS system. Refer to the Power Specifications (page 10) section for specific power capabilities.

The switched voltage lines are turned on and off under program control by the controlling CPU. The switched voltages are:

- +5.7VSW
- +VHIGH
- +VLOW
- −VLOW

Two of the switched voltages (+VHIGH and +VLOW) are variable. The +VLOW line provides +8V to +15V at 700 mA as determined by a precision resistance on the +VLSEL line. The +VHIGH line provides +8V to +27V at 300 mA as determined by a precision resistance on the +VHSEL line.

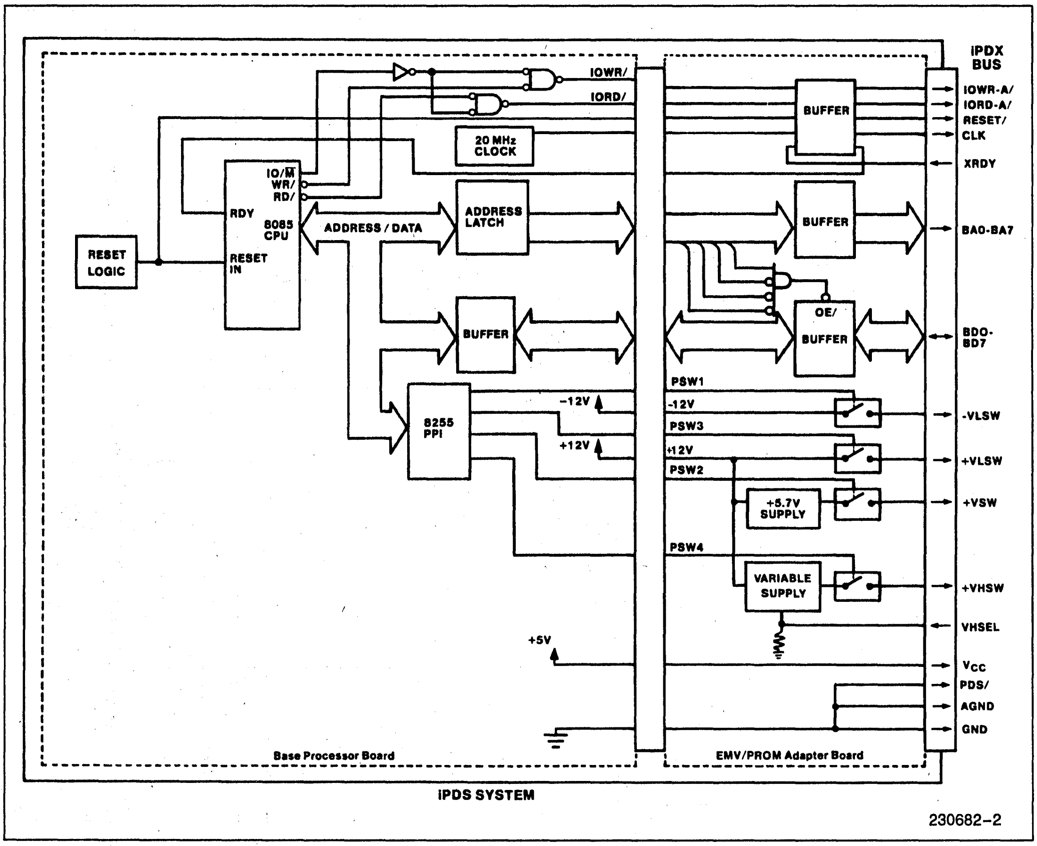


Figure 2. IPDS™/iPDX Bus, Functional Block Diagram

Refer to the Power Considerations (page 14) section for details on the control of the variable supply voltages.

The iUP/IPDX bus implementation provides not only program control of the switched voltage lines. It also allows monitoring of the on/off condition of these lines. The I/O ports used to control and monitor the switched voltages are discussed in the Switched Voltage Programming section (page 22).

Buffered data (AD0–AD7) is placed on the iPDX bus each time address line 4 (A4) is '1' during I/O accesses by the controlling CPU. This ensures that the data lines will be active for I/O addresses of 10H to 1FH. It also places data on the bus for addresses of 3XH, 5XH, 7XH, 9XH, BXH, DXH and FXH. The iPPS software only uses I/O addresses of 1XH when initially contacting the plug-in module, so there is no problem with this I/O addressing.

The address, read, write, reset and ready lines feed directly from the iUP system to the plug-in module on the iPDX bus. Figure 1 is a functional block diagram of the iUP system that shows the iPDX signals, their direction of flow, and the controlling circuitry in the iUP system. Refer to other sections of this application note for specific details on iUP/iPDX bus implementation.

iPDS™/iPDX BUS IMPLEMENTATION

The iPDS system implementation of the iPDX bus is a powerful, general-purpose interface to plug-in modules. The iPDS interface has less power handling capabilities than the iUP interface, but it has additional system resources.

The iPDX/iPDX bus interface uses a separate board in the iPDS system. The iPDS-140 option for the iPDS system is an interface between the iPDX bus and the base processor board of the iPDS system. The iPDS-140 option buffers all address, data, and control signals that go to the iPDX bus. The top address nibble is decoded on the iPDS-140 option to enable data transfers during reads or writes to I/O addresses 10H to 1FH.

The switched voltages for the iPDX bus are developed on the iPDS-140 option. The iPDS-140 option uses +12VDC and −12VDC from the iPDS system to generate the switched voltages. Refer to the Power Specifications and Power Considerations sections (pages 10 and 14) for details on the power available for the iPDX bus.

The switched voltages are under program control of the CPU in the iPDS system. These control signals are sent through an 8255 PPI chip to the iPDS-140 option. Refer to the Programming Switched Voltages section (page 22) for details on switched voltage control.

The V_{CC} (+5VDC) and ground lines from the base processor board are fed directly to the iPDX bus. The PDS/ and AGND lines of the iPDX bus are connected to the ground line within the iPDS-140 option. The PDS/ line is used by PROM programming plug-in modules to indicate the controlling system to iPPS software. All PROM programming plug-in modules feed the PDS/ line (J1–20) back so iPPS software can read its '1' or '0' status. Refer to the iPPS Software Protocol section (page 14) for details on the module status byte.

The address, read, write, reset, clock, and ready lines are buffered on the iPDS-140 option, but they are not modified by the iPDS system. Figure 2 is a functional block diagram of the iPDS system that shows the iPDX signals, their direction of flow, and the controlling circuitry in the iPDS system. Refer to other sections of this application note for specific details on iPDS/iPDX bus implementation.

iPDX BUS SPECIFICATIONS

The specifications for the iPDX bus are divided into four categories:

- Signal listings and descriptions.
- Detailed power (DC) specifications.
- Detailed timing (AC) specifications.
- Outline drawings and detailed mechanical specifications.

iPDX Bus Signal Descriptions

Table 3 presents the pinout of the iPDX bus and gives the associated signal names for both the iPDS and iUP systems.

Table 4 lists the signal names (iPDS and iUP systems) of the iPDX bus and gives a short description of each group of signals.

Table 3. iPDX Bus Pinout

Pin	iPDS Mnemonic	iUP Mnemonic	Input/ Output	Pin	iPDS™ Mnemonic	iUP Mnemonic	Input Output
1	GND	GND	O	22	GND	GND	O
2	GND	GND	O	23	Reserved	Reserved	N/A
3	BA0	AA0	O	24	BD0	AD0	I/O
4	BA1	AA1	O	25	BD1	AD1	I/O
5	BA2	AA2	O	26	BD2	AD2	I/O
6	BA3	AA3	O	27	BD3	AD3	I/O
7	BA4	AA4	O	28	BD4	AD4	I/O
8	BA5	AA5	O	29	BD5	AD5	I/O
9	BA6	AA6	O	30	BD6	AD6	I/O
10	BA7	AA7	O	31	BD7	AD7	I/O
11	V_{CC}	+5V	O	32	Reserved	Reserved	N/A
12	V_{CC}	+5V	O	33	+VHSW	+VHIGH	O
13	+VSW	+5.7VSW	O	34	+VLSW	+VLOW	O
14	+VSW	+5.7VSW	O	35	Reserved	Reserved	N/A
15	CLK	Not Used	O	36	−VLSW	−VLOW	O
16	IOWR-A/	AIOWRT/	O	37	AGND	AGND	O
17	IORD-A/	AIORD/	O	38	+VHSEL	+VHSEL	I
18	RESET/	ARST/	O	39	Not Used	+VLSEL	I
19	XRDY	ARDY	I	40	GND	GND	O
20	PDS/	PDS/	O(iPDS)	41	GND	GND	O
21	GND	GND	O				

Table 4. iPDX Bus Signal Descriptions

Signal Name(s)		Description
iPDS™	iUP	
GND	GND	Reference potential for all signals and supply voltages.
AGND	AGND	Analog ground. Reference potential for the programmable high voltage signal (+VHSW or +VHIGH).
BA0–BA7	AA0–AA7	Address lines from the iPDS system or the iUP system that define the I/O register to be accessed
BD0–BD7	AD0–AD7	Bi-directional, parallel data lines between the plug-in module, and the iPDS or the iUP system.
V_{CC}	+5V	Supply voltage for plug-in module circuitry.
CLK	Not Used	Clock signal (20 MHz) from the iPDS system.
IOWR-A/	AIOWRT/	I/O write signal from the iPDS or the iUP system. An active low indicates that output data from the iPDS or the iUP system is on the data lines. Data is sampled on the trailing edge of this signal.
IORD-A/	AIORD/	I/O read signal from the iPDS or the iUP system. An active low indicates that input data from the plug-in module should be placed on the data lines. Data is sampled on the trailing edge of this signal.
RESET/	ARST/	Reset signal from the iPDS or the iUP system.
XRDY	ARDY	Asynchronous ready signal from the plug-in module. An active high indicates that the plug-in module has accepted write data from, or presented valid read data to, the iPDS or the iUP system. A low level causes the iPDS or the iUP system to enter a wait state after either the IORD-A/ (AIORD/) or IOWR-A/ (AIOWRT/) line is activated.
PDS/	Not connected	A ground from the iPDS system. This signal is sampled by iPPS software and indicates that a PROM programming module is installed in an iPDS system.
+VSW	+5.7VSW	Switched +5.7VDC that can be turned on or off by the iPDS or the iUP system under program control.
+VHSW	+VHIGH	Switched +8VDC to +26VDC that can be turned on or off by the iPDS or the iUP system under program control. The actual voltage is determined by the +VHSEL signal from the plug-in module.
+VLSW	+VLOW	Switched +8VDC to +13VDC that can be turned on or off by the iPDS or the iUP system under program control For the iUP system the actual voltage is determined by the +VLSEL signal from the plug-in module. The iPDS system outputs only a fixed voltage of +12VDC on the +VLSW line.
−VLSW	−VLOW	Switched −12VDC that can be turned on or off by the iPDS or the iUP system under program control.
+VHSEL	+VHSEL	High plus programming voltage select (iPDS and iUP systems). A precision resistance in the plug-in module determines the voltage on the +VHSW (+VHIGH) line.
Not Used	+VLSEL	Low plus programming voltage select (iUP system only). A precision resistance in the plug-in module determines the voltage on the +VLOW line.

Power Specifications

The +5VDC line is always active on the iPDX bus. This line normally powers plug-in module circuitry. Switched voltages are also available to power plug-in module circuitry. The user must first set up appropriate driver routines and programming voltages before the switched voltage lines become active.

Table 5 lists the supply signals available at the iPDX bus and the specifications for each signal.

Figure 3 shows the power available on the iPDS +VHSW signal line for the programmable voltages. The other power supply signals give rated power over their full range.

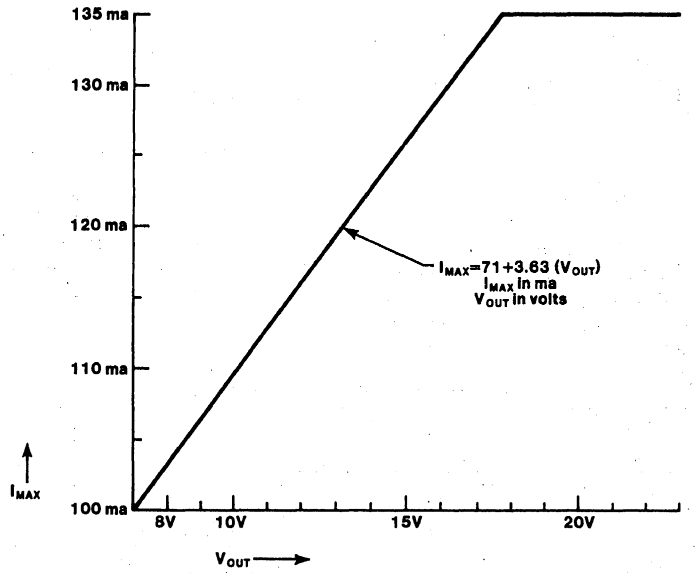


Figure 3. Power Available (iPDS™ +VHSW Signal)

Table 5. iPDX Bus Power Specifications

Signal Name		Supply Voltage and Tolerance		Maximum Current		Notes
iPDS™	iUP	iPDS	iUP	iPDS	iUP	
V_{CC}	+5V	+5VDC ±2.5%		2.5 amps	1.0 amps	
+VSW	+5.7VSW	+5.7VDC ±50 mv		250 mA	1.5 amps	1
+VHSW	+VHIGH	+8VDC to +27VDC ±2%		135 mA	300 mA	1, 2
+VLSW	+VLOW	+12VDC ±1.0V	+8VDC to +15VDC ±2%	200 mA	700 mA	1, 3
−VLSW	−VLOW	−12VDC ±0.5V		50 mA	100 mA	1

NOTES:
1. This voltage is switched and is under program control of the iPDS or the iUP system.
2. The voltage is controlled by the +VHSEL signal. Figure 3 shows the derating required for each selected voltage of +VHSW.
3. The voltage is controlled by the +VLSEL signal (iUP system only).

Electrical (DC) Specifications

The signal names for the iPDX bus indicate whether or not the signals are active high or active low. If the name ends with a slash (/), the signal is active low. If the name has no slash following it, the signal is active high. Table 6 shows the electrical specifications for the iPDX bus.

The electrical characteristics for the iPDX bus signals are shown in Table 7. The voltage and current specifications refer to the TTL high or TTL low state of the iPDX bus signal. The signal type (input or output) is the signal direction when viewed from the iPDS or the iUP system side of the iPDX bus. Positive currents are defined as currents entering the interface.

Timing (AC) Specifications

Figure 4 shows the timing specifications for the iPDX bus. Table 8 lists definitions of the timing parameters used for the iPDX bus. Refer to the *MCS®-80/85 Family User's Manual* or the *8085A-2* data sheet for specific details on the timing specifications for the iPDX bus.

The +VHSEL/+VLSEL signals, the data bus signals, and the ready (XRDY or ARDY) signal originate in the plug-in module. The voltage select and data bus signals have straightforward timing requirements, but the timing requirements for the ready signal need explanation.

CAUTION

230682-4

Whenever the ready signal (XRDY or ARDY) goes low, the CPU generates wait-states until the ready signal returns high. The ready signal should not be driven low for more than a few bus cycles unless complete suspension of all CPU bus activity is allowable in the user's application.

The ready signal is normally high for all read/write transfers over the iPDX bus. The ready signal can be driven low to insert one or more wait-states in the CPU bus cycle, in cases where the plug-in module uses slow memory devices or slow peripheral devices.

Table 6. iPDX Bus Electrical Specifications

Active State	Logical State	Electrical Signal Level	At Receiver	At Driver
LOW	0	H = TTL High State	5.25V ≥ H ≥ 2.0V	5.25V ≥ H ≥ 2.4V
	1	L = TTL Low State	0.8V ≥ L ≥ −0.5V	0.5V ≥ L ≥ 0.0V
HIGH	0	L = TTL Low State	0.8V ≥ L ≥ −0.5V	0.5V ≥ L ≥ 0.0V
	1	H = TTL High State	5.25V ≥ H ≥ 2.0V	5.25V ≥ H ≥ 2.4V

Table 7. Electrical Characteristics of iPDX Bus Signals

Signal Type	I_{OL} Max	I_{IL} Max	I_{OH} Max	I_{IH} Max	V_{OL} Max	V_{IL} Max	V_{OH} Min	V_{IH} Min
All Outputs	24 mA		−5 mA		0.5V		2.4V	
Inputs (except RDY signal)		−12.8 mA		50 μA		0.8V		2.0V
ARDY (input)		−4 mA		50 μA		0.8V		2.0V

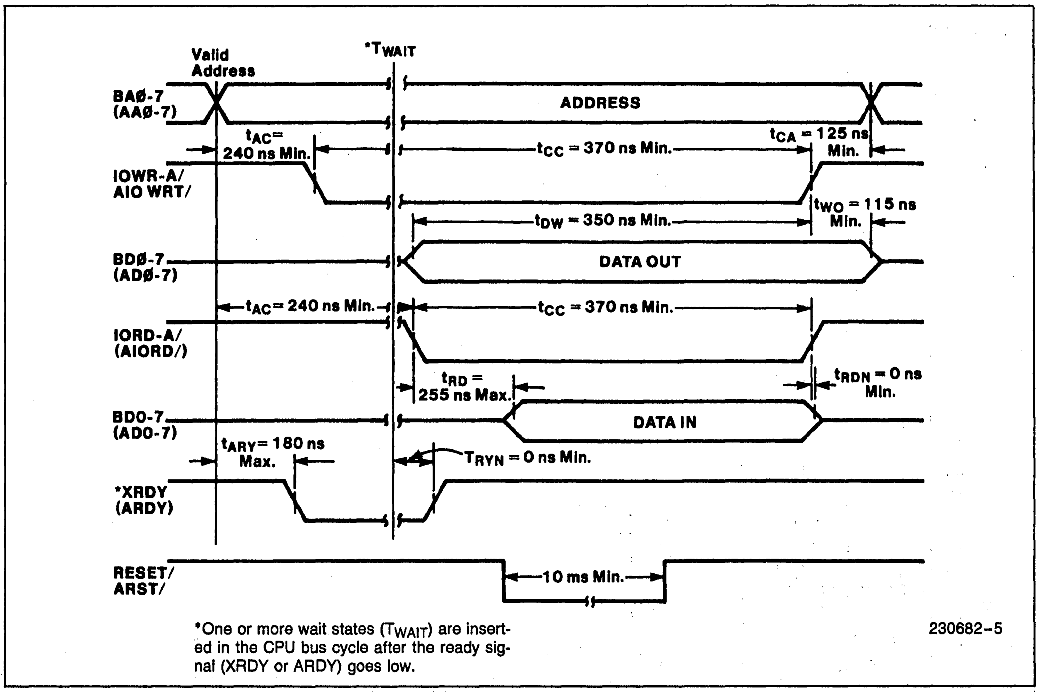


Figure 4. iPDS Bus Read/Write Timing

Table 8. iPDX Timing Definitions

Symbol	Description
t_{AC}	The time between valid address (A0–A7) and the leading edge of the control signal.
t_{ARY}	The time between valid address (A0–A7) and the trailing edge of the ready signal.
t_{CA}	The time between the trailing edge of the control signal and the end of valid address.
t_{CC}	The width of the control signal.
t_{DW}	The time between the start of valid data (D0–D7) and the trailing edge fo the write control signal.
t_{RD}	The time between the leading edge of the read control signal and the start of valid data (D0–D7).
t_{RDH}	The time between the trailing edge of the read control signal and the end of valid data (D0–D7).
t_{RYH}	The time between the end of T_{WAIT} and the leading edge of the ready signal.
t_{WD}	The time between the trialing edge of the write control signal and the end of valid data (D0–D7).

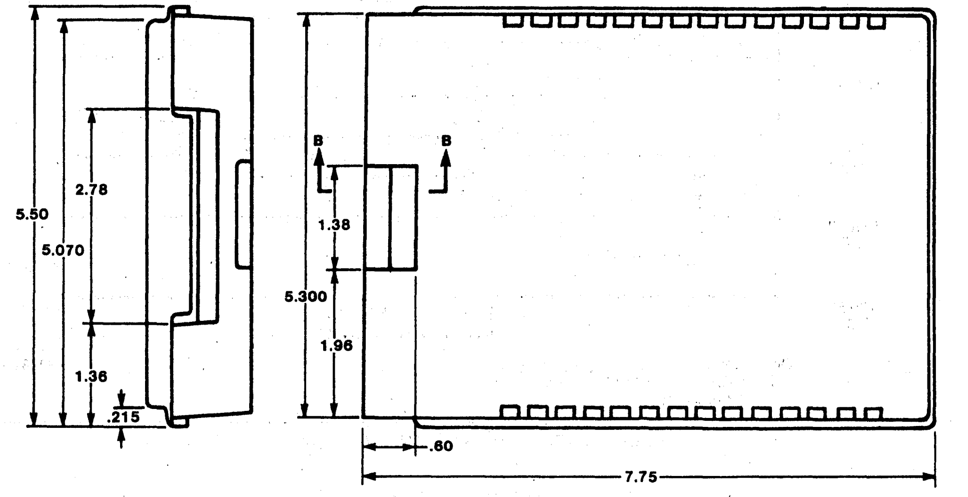
Mechanical Specifications

The mechanical specifications define the connector requirements and the outline and mounting dimensions for plug-in modules using the iPDX bus. Figure 5 is an outline drawing of a plug-in module for the iPDX bus. All plug-in modules for the iPDX bus must comply with the dimensions specified in Figure 5.

HARDWARE DESIGN CONSIDERATIONS

Plug-in modules designed around the iPDX bus must follow certain design rules. These design rules are:

- The first four inches (measured from the connector end) of the plug-in module must meet the mechanical and outline specifications shown in Figure 5.



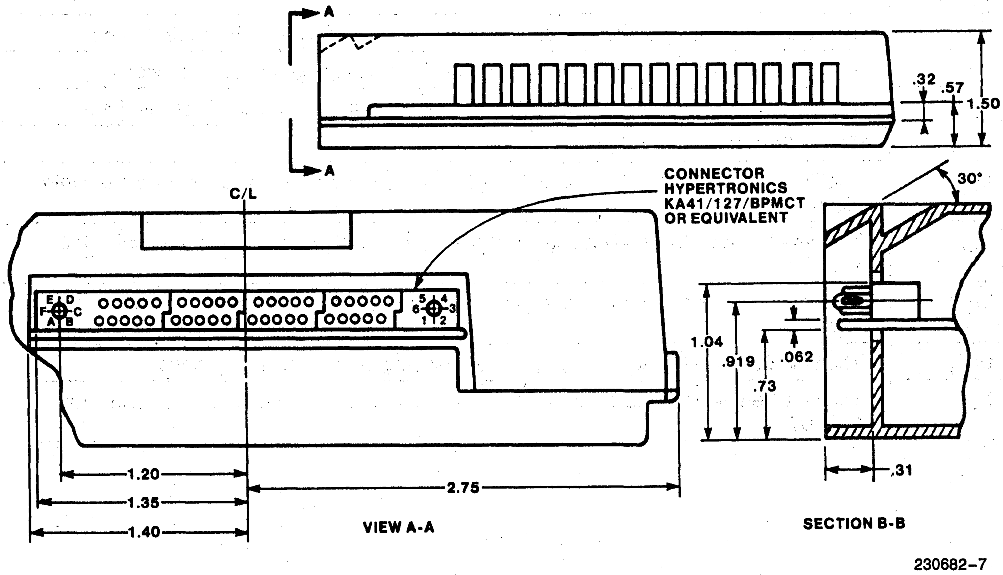


Figure 5. Plug-In Module Mechanical Specifications

- The maximum V_{CC} (+5VDC) current available is 2.5 amps for iPDS plug-in modules or 1.0 amps for iUP plug-in modules.
- Switched voltages of +5.7VDC, +8VDC to +27VDC, +12VDC and −12VDC are available to circuitry on a plug-in module under program control Table 5 lists power specifications for the iPDX bus.
- If a programmed voltage (positive only) is required by the plug-in module, an appropriate precision resistor must be installed in the plug-in module.
- All signals (except +VHSEL and +VLSEL) returned by the plug-in module must be TTL levels.
- Provisions must be made to sample the PDS/ signal on PROM programming plug-in modules that use iPPS software while connected to an iPDS system. (The PDS/signal is low when the module is connected to the iPDS system and floating when connected to the iUP system. Firmware can use the signal 1) to specify whether a power supply status port is available, 2) to specify whether E3H (iPDS) or 03H (iUP) is the correct port for turning on power supplies, and 3) to compensate for differences in timing between the two systems.)
- Direct memory access (DMA) transactions are not supported on the iPDX bus.

Mechanical Considerations

Plug-in modules for the iPDX bus must have an enclosure that meets the mechanical specifications shown in Figure 5 for the first four inches (measured from the connector end) of the module. Intel has developed a prototyping kit (product code iPDS-PROTO) to simplify the mechanical and hardware portions of the design. This prototyping kit consists of the plug-in module enclosure, a prototyping board, iPDX bus connector, a hardware kit, isolation capacitors, and wire-wrap pins. The iPDS-PROTO kit can accept up to 30 ICs and associated discrete components in the available board space. If a plug-in module designed around the iPDX bus goes to a production phase, use of the module tooling can be licensed through Intel.

Power Considerations

The maximum power dissipation for an iPDS plug-in module is 20.5 watts with a maximum draw of 12.5 watts from the V_{CC} line. The maximum power dissipation for an iUP plug-in module is 32.5 watts with a maximum draw of 5.13 watts from the V_{CC} line and 8.625 watts from the +5.7 VSW line. A maximum of 7.5 watts can be dissipated within a plastic plug-in module (more power can be dissipated at the PROM socket).

V_{CC} (+5VDC) is the only voltage present at all times on the iPDX bus. If the plug-in module circuitry requires other voltage levels for operation, the switched voltages must be turned on first by software. The Programming Considerations section shows the iPDX bus set-up requirements for turning on/off each of the switched voltage signals.

The variable switched voltages (+VHSW on an iPDS plug-in module, and +VHIGH and +VLOW on an iUP plug-in module) use one or more precision resistors on the plug-in module to determine their line voltage. The precision resistor on the plug-in module must be connected between the AGND line and the +VHSEL line of the iPDX bus. Plug-in modules for an iUP system can also program the +VLOW line by connecting a precision resistor between the AGND line and the +VLSEL line of the iPDX bus. Figure 6 shows a chart and two equations that indicate the precision resistor values corresponding to programmable voltages. Figure 7 shows three kinds of circuits that allow the plug-in module to select more than one programming voltage level.

PROGRAMMING CONSIDERATIONS

PROM programming modules are normally controlled by iPPS software residing in either the iPDS or the iUP system. User-designed plug-in modules (other than programming plug-in modules) are controlled by user-supplied driver programs. The iPPS Software Protocol section explains the iPPS-iPDX bus interface. The Switched Voltage Programming section gives programming requirements for accessing switched voltages in the iPDS/iPDX bus interface. The User-Written iPDX Bus Drivers section presents the programming requirements for user-supplied driver programs.

iPPS Software Protocol

PROM programming plug-in modules that run under control of iPPS software must contain firmware. The firmware in the PROM programming module is a program that has routines for programming the device(s) that the plug-in module is designed to program. This firmware is uploaded into RAM in the controlling (iPDS or iUP) system the first time the TYPE command in the iPPS command language is executed. After the module firmware is uploaded, the iPDS or the iUP system controls the programming operation. The iPPS software communicates with the plug-in module over 7 of the 16 I/O ports allocated for iPDX bus communication. Table 9 lists the I/O port assignments recognized by iPPS software.

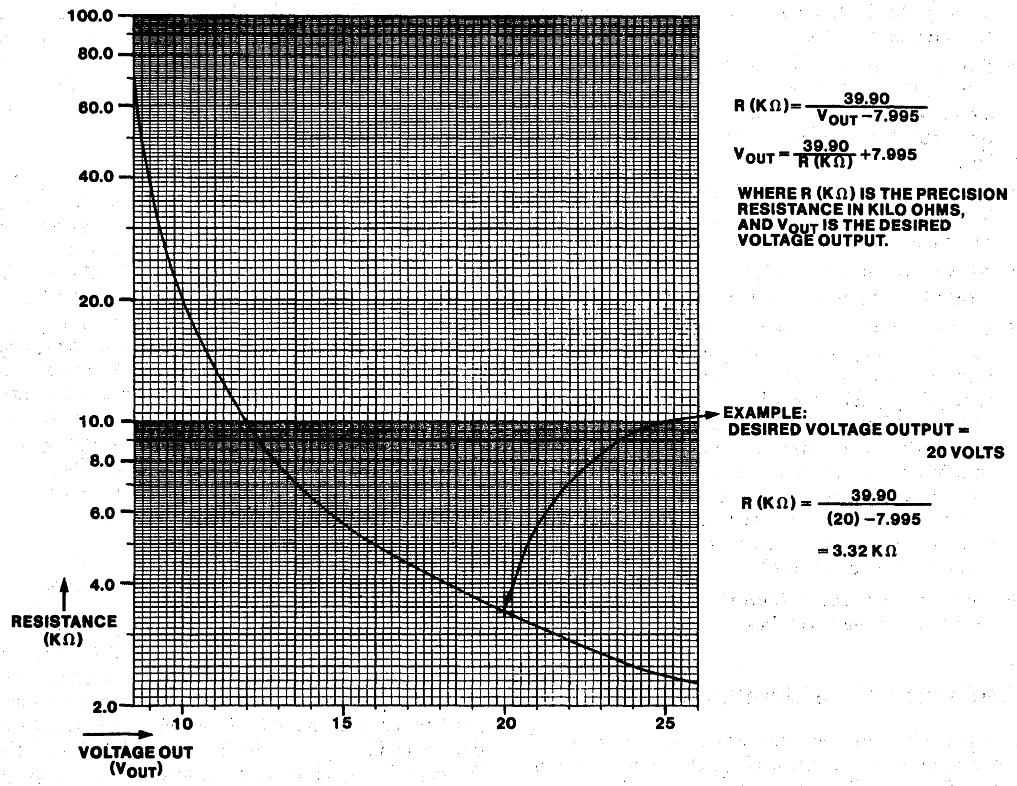


Figure 6. Programmable Voltage Resistor Values

The control words, corresponding to an I/O write to port addresses 10H, 11H and 12H, control various functions on the plug-in module. These functions may include voltage select and routing for the target PROM socket, the programming pulse, or chip selects, and set/clear the upload flag. The bit definitions for the control words are shown in Figure 8.

The status word, corresponding to an I/O read of port address 10H, contains information about the current state of monitored functions on the plug-in module. The bit definitions for the status word are shown in Figure 9.

The plug-in module firmware is read when the iPPS TYPE command is first executed. The iPPS software uploads plug-in module firmware by writing the plug-in module PROM location to I/O ports 13H (A0–A7) and 14H (A8–A15), respectively, and then reading the data at I/O port 11H. The plug-in module firmware uploads to absolute address 7020H in the iPDS or iUP system. After the plug-in module firmware is uploaded to the iPDS or the iUP system, the upload flag (bit 1 of control word 0) is set by the controlling system. Setting the upload flag causes bit 1 of the status word to indicate that additional firmware uploads are not required.

PLUG-IN MODULE FIRMWARE

The firmware (for plug-in modules running under control of iPPS software) controls all plug-in module operation, except the firmware upload operation itself. This firmware must be written in 8085 code and formatted as shown in Table 10.

The first two bytes of plug-in module firmware must contain the total number of bytes to be uploaded (including the two length bytes and the two check-sum bytes). The third byte must contain the number of different devices the plug-in module can read or program.

The plug-in module firmware is divided into segments and a segment is required for each PROM type that the module can program. Each segment contains a descriptor (first 14 bytes) and a code section.

Descriptor Section

The first two descriptor bytes contain the address of the next segment of firmware. The last segment of the firmware must contain the address of the first segment. If there is only one segment, the segment must reference itself.

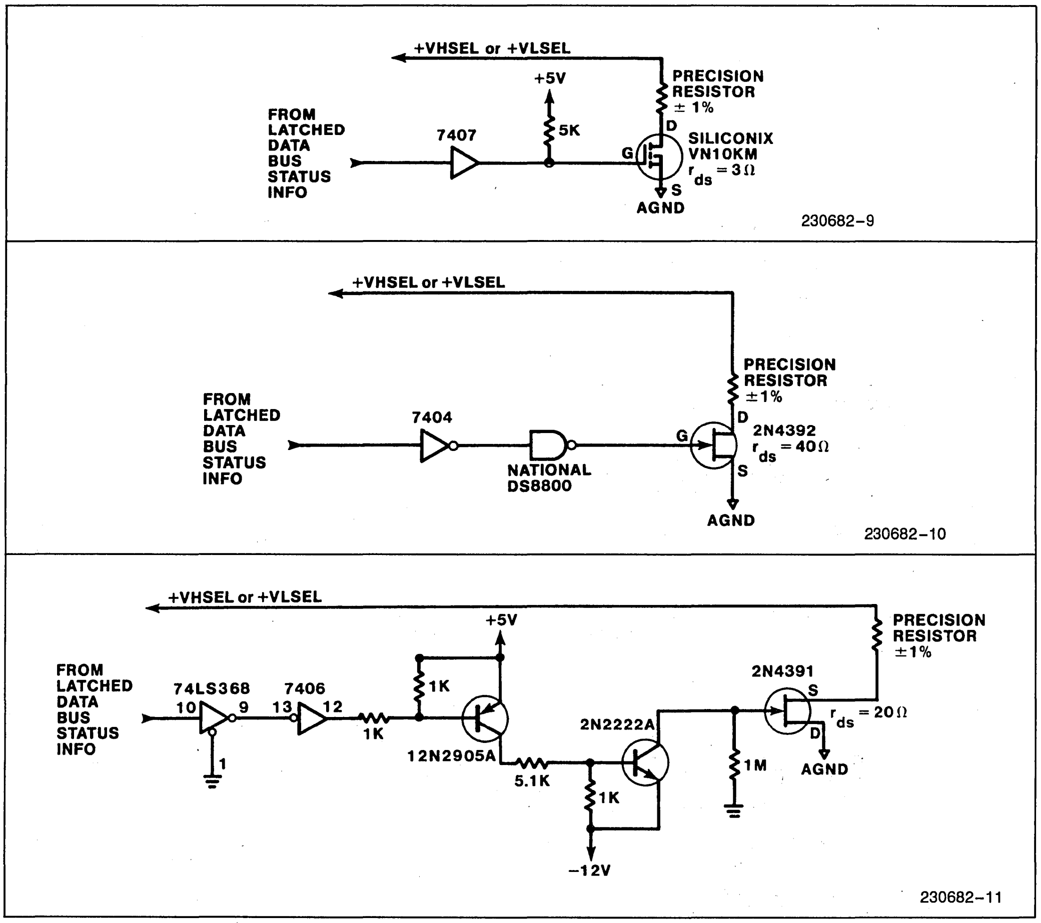


Figure 7. Three Precision Resistor Switching Circuits

Table 9. I/O Port Assignments Used by iPPS Software

I/O Port Address	I/O Write Active	I/O Read Active
10H	Write control word 0	Read module status
11H	Write control word 1	Read personality PROM data
12H	Write control word 2	Available
13H	Write address (A0–A7)	Available
14H	Write address (A8–A15)	Available
15H	Write address (A16–A19)	Available
16H	Write data (D0–D7)	Read device data

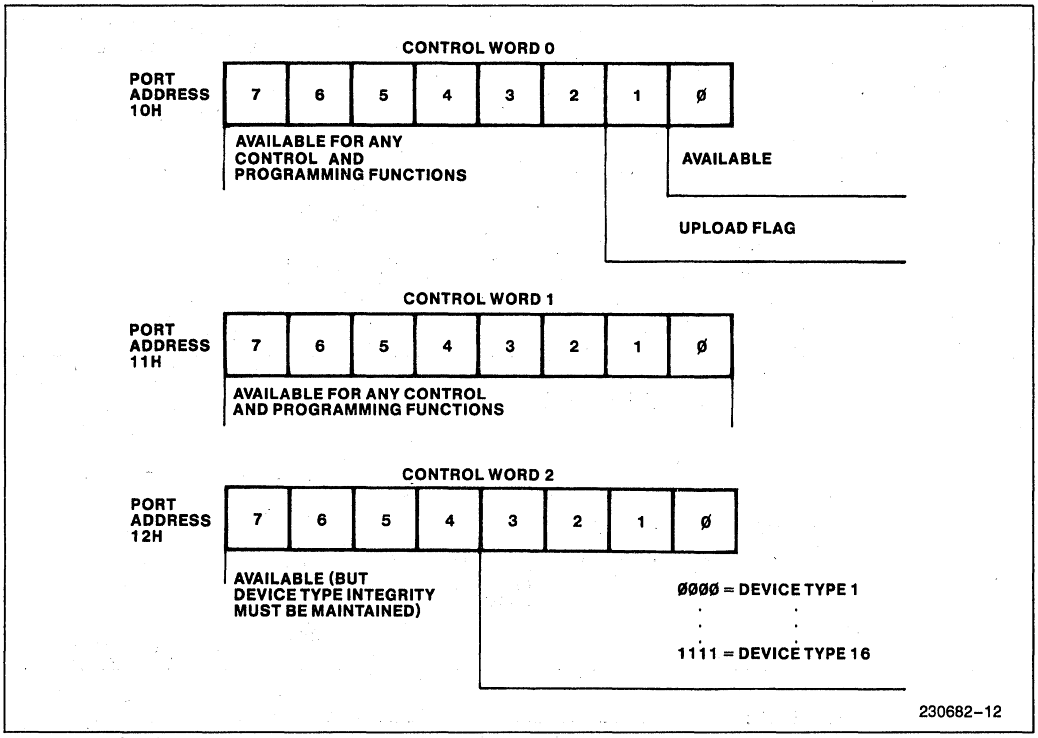


Figure 8. iPPS Control Word Bit Definitions

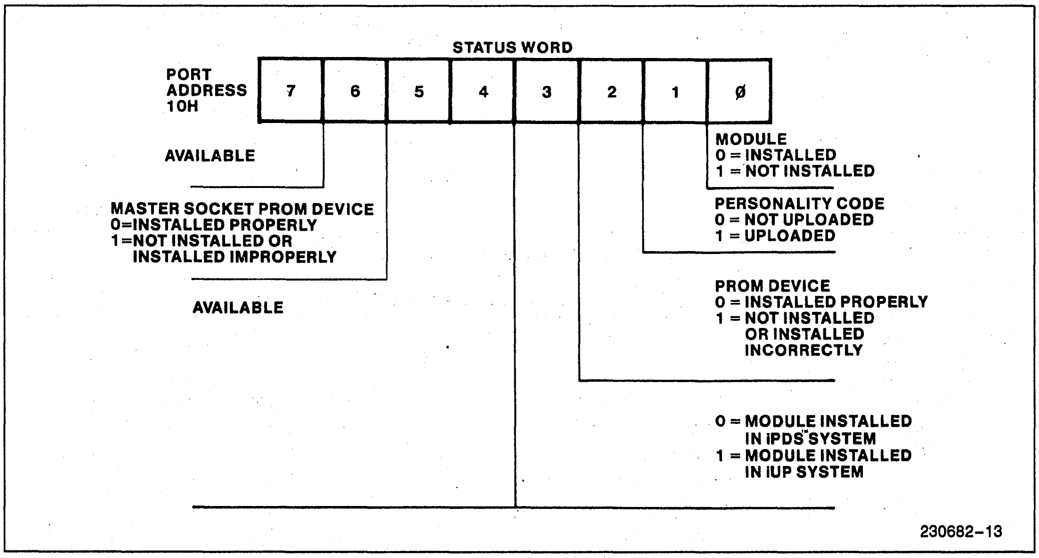

STATUS WORD
PORT ADDRESS 10H
7 6 5 4 3 2 1 Ø
AVAILABLE
MASTER SOCKET PROM DEVICE 0=INSTALLED PROPERLY 1=NOT INSTALLED OR INSTALLED IMPROPERLY
AVAILABLE
MODULE 0 = INSTALLED 1 = NOT INSTALLED
PERSONALITY CODE 0 = NOT UPLOADED 1 = UPLOADED
PROM DEVICE 0 = INSTALLED PROPERLY 1 = NOT INSTALLED OR INSTALLED INCORRECTLY
0 = MODULE INSTALLED IN iPDS SYSTEM 1 = MODULE INSTALLED IN iUP SYSTEM
230682-13

Figure 9. iPPS Status Word Bit Definitions

Table 10. Plug-In Module Firmware Format

Personality Prom Address			Contents
		0	8 LSBS of the length of the personality PROM.
		1	8 MSBS of the length of the personality PROM.
		2	Number of types the module can program.
		3	8 LSBS of the address of the next segment in the table (U).
	D	4	8 MSBS of the address of the next segment in the table (U).
	E	5	1st ASCII character of PROM type.
	S	6	2nd ASCII character of PROM type.
	C	7	3rd ASCII character of PROM type.
	R	8	4th ASCII character of PROM type.
	I	9	5th ASCII character of PROM type.
	P	10	6th ASCII character of PROM type.
	T	11	7th ASCII character of PROM type.
	O	12	8th ASCII character of PROM type.
	R	13	8 LSBS of PROM address range.
		14	8 MOBS of PROM address range.
S		15	8 MSBS of PROM address range.
		16	Bits 0–5 indicate PROM word length. Bit 6 indicates the blank state of the PROM.
E			Bit 7 is not used
		17	Jump to blankcheck routine (V).
G		18	8 LSB of address of blankcheck routine.
		19	8 MSB of address of blankcheck routine.
M		20	Jump to program routine (W).
		21	8 LSB of address of program routine.
E		22	8 MSB of address of program routine.
		23	Jump to overlay check routine (X).
N		24	8 LSB of address of overlay check routine.
	C	25	8 MSB of address of overlay check routine.
T		26	Jump to reverse socket routine (Y).
	O	27	8 LSB of address of reverse socket routine.
		28	8 MSB of address of reverse socket routine.
	D	29	Jump to read routine (Z).
		30	8 LSB of address of read routine.
	E	31	8 MSB of address of read routine.
		V	Start blankcheck code.
		V+N	"RETURN"
		W	Start code for program routine.
		W+N	"RETURN"
		X	Start code for overlay check.
		X+N	"RETURN"
		Y	Start code for reverse socket routine.
		Y+N	"RETURN"
		Z	Start code for read routine.
		Z+N	"RETURN"
Next segment (U)			
Next two locations after last byte of last segment			Checksum (LSB) Checksum (MSB)

The next eight descriptor bytes contain the ASCII code for the device being programmed. Spaces (ASCII code 20H) must be used to fill any unused bytes of this ASCII code.

The remaining four descriptor bytes contain specific PROM device information, with the first three bytes holding the available PROM address range and the final byte holding PROM data information. Bits 0–5 of the PROM data information byte contain the word length (binary equivalent in bits) of the selected PROM. Bit 6 of the PROM data information byte indicates the unprogrammed state of each PROM bit (i.e., a 0 in the bit 6 location means a device bit is unprogrammed in the high state and programmed in the low state). Bit 7 of the PROM data information byte is not used.

Code and Checksum Sections

The code section is subdivided into a jump op code section followed by blankcheck, program, overlay check, reverse socket detect, and read routines.

The jump op code section contains the jump op codes and addresses of each programming routine for the device covered in this segment. The programming routines referenced in this section include read, blankcheck, program, overlay check, reverse socket detect, and read. The referenced routines may actually reside in other segments.

The blankcheck, program, overlay check, reverse socket detect, and read programming routines must be in 8085 code. These routines are hardware specific instructions for checking and programming the device. The following subsections describe relevant details of these routines and provide other information needed to develop module firmware.

The final two bytes of firmware following the last segment contain the checksum for the plug-in module firmware chip. The checksum is the 2's complement of the sum of the previous bytes in the plug-in module firmware chip.

Memory Variable and Stack Locations—Memory locations 6000H to 60FFH are reserved for variables and stack. Please note that this leaves space for a very small stack. The following is a list of variables that the user needs to know to interface to iPPS software.

6000H	Lowest address for 80 bytes of input buffer.
6050H	Lowest address for 80 bytes of output buffer; space is also used for variables when PROMs greater than 32K bytes are edited.
601AH	Used to indicate on-line (00H) or off-line (01H) operation.
60A2H	Used to pass the current status of the iUP programmer to the iPPS software.
60B4H-60B5H	Both 60B4H and 60B5H are general purpose locations for passing information. See information in this section on creating firmware for displaying messages on the host.
60B6H	Used to indicate when powering down has finished, i.e., when an operation has been completed. The module firmware should set this location to 01H when power is turned on. This location is reset to 00H when the power is shut off. This information is needed by the iPDS system, since the iPDS system does not have a status port (such as 02H in the iUP programmer) to indicate whether power is on or off.
60B7H	For passing an address between module and iPPS software: contains LSB of address.
60B8H	For passing an address between module and iPPS software: contains MOB of address.
60B9H	For passing an address between module and iPPS software: contains HOB of address.
60BAH	Contains data to be programmed from the iUP programmer to PROM.
60BBH	Contains data read from PROM to iUP programmer.
60CCH	Indicates operation in process. Used in off-line keyboard interrupts. See keyboard interrupt routine below.
60CFH	Used for the lock function. The iPPS software sets this location to 00H before calling the reverse socket check. The module firmware sets this location to FFH if a lock function is available or leaves it at 00H to indicate that no function is available. (This ensures backwards compatibility with older modules.) The iPPS software then sets this location to 01 before calling the programming routine. This value indicates to the module that lock (rather than programming) is requested. (If programming is requested, the value is 00H.)
60D0H	Used in the lock function. The module firmware uses this location to indicate which parameter is being passed. On modules that just lock (like 8751AH), the lock sequence will never go above 1.

On authenticated PROMs, the sequence numbers may be greater than 1. This allows the module, iPPS software, or user to edit the parameters. The parameters should be stored in a buffer and this location is used to index the buffer. If the user responds NO to the EXECUTE query, module firmware should reset this location to the beginning (0). The buffer values (instead of the PROM's actual values) are then sent back. These locations are programmed only when the user responds YES to the EXECUTE query. Module firmware should be set to 0 when finished.

60D2H Indicates a PROM that is greater than 32K bytes has been edited. (00H = NO; 01H = YES).

60D3H Indicates whether the module should be using the programming socket. There is a bug in the initialization of this flag, so until iPPS-PDS software and the iUP programmer firmware are upgraded, the module firmware needs to set this location as follows:

(1) For PROMs less than 32K bytes, set to 00.

(2) For all devices when on-line, set to 00.

This covers the two conditions in which the master socket will never be accessed.

60FFH Top of the stack.

Parameters for Major Subroutines—Unless otherwise noted, the module returns results using the following codes:

00H means "pass".
12H means "power supply failure".
07H means "abort".

Information on Code Section Routines—The following paragraphs provide information on routines included in the code section of the PROM programming firmware. Note that the meaning of "iUP programmer" in these paragraphs depends on the system being considered. "iUP programmer" can mean either iUP-200A/201A firmware or iPPS-PDS software.

Blank Check Routine—The iUP programmer passes no parameters to the module. The module firmware checks the entire PROM and passes back results in the B register. (Fail = 05H.) If the PROM fails the blank check test, the actual value of the PROM is passed back in 60BBH and the location in 60B7H, 60B8H, and 60B9H. In the off-line mode, any undefined value in B defaults to abort.

Program Routine—The iUP programmer sends the location to be programmed in 60B7H, 60B8H, and 60B9H, and sends the data to be programmed in 50BAH. It also resets 60CFH to 00H. The module returns results in the A register. (Fail = 01.) In the off-line mode, any undefined results default to abort. If the programming failed, the actual value of the PROM is passed back in 60BBH and the location in 60B7H, 60B8H, and 60B9H. The off-line error message will show the address of the failure and user data XOR PROM data. In the on-line mode, the host console will the show failure address, user data, and PROM data.

Overlay Check Routine—The iUP programmer passes no parameters. The iPPS software does not use the overlay check routine; it does its own overlay check on the portion of PROM to be programmed.

In the off-line mode, data the user wants to program is in memory starting at 8000H, and the entire PROM is checked with results sent back in the B register. (Fail = 01.) The module firmware may also send back 03H in the B register to indicate that the iUP programmer should perform the overlay check (on edited PROMs greater than 32K, the iUP programmer automatically performs the overlay check). Any undefined result defaults to abort.

The iUP programmer uses the following algorithms to determine whether the new user data can be programmed over a nonblank PROM location:

1. For PROMs with FFH as a blank state:

   IF [(user data AND PROM data) XOR user data = 0] THEN overlay is possible

2. For PROMs with 00H as a blank state:

   IF [(user data XOR PROM data) AND PROM data = 0] THEN overlay is possible

Reverse Socket Check Routine—The iUP programmer indicates in 60D3H which socket to check and initializes 60CFH to 00H. The module sends back results in the A register. (Fail = 04H.) In the off-line mode, the iUP programmer only recognizes pass, abort, and will default to fail for any other unrecognized result. On chips which support the lock function, 60CFH is set to FFH; on old modules or for chips that do not support the lock function, 60CFH is left at 00H. Addition of other initialization tests can be accomplished by adding these tests to the module reverse socket code. Then, if an error occurs, the module can send a specific error message and abort.

Read Routine—The iUP programmer passes the location to be read in 69B7H, 60B8H, and 60B9H; a code for the (master or program) socket that is to be read from is passed in SKTFLG. The module passes the data read in 60BBH and the result in the A register.

NOTE:
There is no failed status, only pass, abort, or power supply failure, In the off-line mode, any undefined result defaults to power supply failure.

Lock Routine—The iUP programmer checks module installation, sets location 60CFH to 00H, and performs the reverse socket test. If 60CFH still equals 00H after the reverse socket check, then the lock function is not available for that module and/or chip. If, however, 60CFH equals 01H after a reverse socket check, then the lock function is available; 60CFH will remain at 01H until the command is finished.

Next (with 60CFH = 01 and 60D0H = 00H), the iUP programmer calls the program subroutine. The module firmware can then communicate with the user by returning (in the A register) one of the values shown in Table 11. When needed, the HL register pair points to the text to be displayed (where the first byte of the message is the length of the message). Handshaking will continue until the result returned is 00H or one of the aborts occurs. (During this process, data sent by the user is contained in location 60BAH and data from the PROM or buffer is contained in 60BBH.) If data values are required, the module stores these values in a buffer (in the module firmware) using 60D0H as an index. No programming or locking is performed until the user has answered YES to the EXECUTE query. At this point, interrupts are disallowed.

Table 11. A-Register Results

Value	Meaning
00H	Pass/done and 60D0H = 00H
02H	Continue and send message pointed to by HL registers
04H	Send execute query to user
07H	Abort (with message)
09H	Lock not available/illegal operation
0AH	Failed; send "PROM BLANK" message
0BH	Failed; send "LOCK FAILED" message
0CH	Failed; send "LOCK FAILED AT" message
0DH	Illegal parameter value
12H	Power supply failure
17H	Abort (without abort message)

Verify—On-line verification is performed by iPPS software using reads. Upon failure, the addresses, user data, and PROM data are displayed. Off-line verification is done by the iUP programmer firmware. Upon failure, the address and user data or PROM data are displayed. The user then has the option of pressing the VERIFY key again to continue verification or pressing the CLEAR key to abort.

Editing PROMS Larger than 32K Bytes—In the off-line mode, editing of PROMs greater than 32K requires a master socket and some special considerations. The iUP programmer has only 32K of image RAM; so, on PROMs greater than 32K, the iUP programmer expects a master PROM in the master socket. The iUP programmer uses this master PROM as the source for programming and overlay checks. (Note that for PROMs larger than 32K bytes, pressing the ROM-to-RAM key does not load data into the URAM. Thus, in using this method of expanding the editing features of the iUP programmer, it is no longer possible to load a 27512 into URAM and then copy URAM to a 27256.)

When the user wishes to edit (off-line) a PROM greater than 32K, data to be edited is copied in 1K blocks to the URAM. (Each 1K block copied always starts on a 1K boundary.) Up to thirty-one 1K blocks can be copied and edited; the last 1K of URAM is not available because this space is needed to manage the editing.

Power-Down Sequence—For current modules, there is an assumption that the module does not need to know when the iPPS software is going to shut off the power supplies; so, the module firmware cannot find this out. For modules that require a certain power-down sequence, there are two possibilities.

- Plan the module to correspond to the iPPS software power-down sequence:
  1. Port 11H is set to 0.
  2. 60B6H is set to 0.
  3. All bits in port 10H are set to 0 except bit 1 (the upload flag), which is not modified.
  4. All power supplies are shut off.
- Module firmware shuts off selected controls in the appropriate order until there is no danger when the iPPS software decides to shut off power supplies. The one check that may be needed is an off-line check. When off-line, the module always checks, reads, or programs the entire PROM—so that if the module is off-line and at the last address, then the iUP programmer will be powering down.

Creating Firmware for Displaying Messages on the Host—To send messages to be displayed by the host, use the following algorithm.

```
Check locations 60A1H to determine whether the
host is the iUP programmer or iPDS system.
If host is the iUP programmer
  Call 7006H to blank the display
  Set HL to 6050H (output buffer)
  Insert a carriage return as the first character
  Fill in the message in the output buffer
  Increase the byte count of the message by 1 (for
  the carriage return) and place the count in the B
  register
  Set HL = 0
  Call 7003
If the host is on-line (i.e., if the host is the iPDS
system)
  Set 60B4H = 21H to indicate message to iPPS
  software
  Fill in the message starting at 6054H (output buff-
  er plus 3)
  Insert a carriage return and linefeed at the end of
  message
  Set B register = message length plus 6
  Call 7000H
```

(7000H and 7003H are actually jump tables to the real address. The jump tables are generated by iPPS software so that updates to iPPS software will be backwards compatible.)

Power Supply Status—There is no status register (02H) to read to tell whether the power supplies have been turned on in the iPDS. Thus, module firmware must monitor 60B6H, if the host is an iPDS. 60B6H is set to 0 upon initialization and when power supplies are turned off. The module firmware must set it to 1 when the power supplies are turned on and set it to 0 when the power supplies are turned off.

WAIT Routine Difference—The 250 microsecond WAIT routines in the iUP programmer and iPDS firmware are inaccurate for short periods of time and do not match each other exactly. (These routines were not revised to ensure backwards compatibility.) For precise timing, the user should write a loop taking into account the differences between the iUP programmer and iPDS clocks.

Use of the E Register—The E register is reserved for use in keyboard interrupts. The module may use the E register if interrupts are first disabled and a known value is restored before re-enabling interrupts. This use of the E register will cause no key presses to be serviced. It is much safer to leave the E register alone.

Keyboard Interrupt Logic—The keyboard interrupt logic is as follows.

```
  Save PSW and HL
  Save the character read in 60C1H
  If the iUP programmer is on-line
    then if key pressed is the on-line key
      then E register = 81H
      else ignore key pressed
    else if key pressed is clear display
      then E register = 88H
     if 60CCH <> 0 /*if operation is process*/
     E register = 80H            /*value key press*/
    Restore PSW and HL
    Return
```

Switched Voltage Programming

There are four switched voltages on the iPDS bus that are turned on or off under program control. The iPDS and iUP systems use different I/O addresses for programming the switched voltages. Under iPPS software, the plug-in module firmware controls the switched voltages. Under user-prepared driver software, separate commands must be included to turn on or off the required switched voltages.

iUP SWITCHED VOLTAGE PROGRAMMING

The iUP system switches the +5.7 VSW, +VLOW, +VHIGH, and −12 VSW supply lines on and off under program control. The controlling program must write twice to I/O port 03H to set/clear and then clock (high to low transition) the switched voltage flip-flops. The first write to I/O port 03H must have bit 0 (clock) high and bits 1 through 4 set for the desired program voltages. The second write to I/O port 03H keeps bits 1 through 4 at the desired program voltage level while bit 0 goes low. The on/off status of each switched voltage line can be checked by reading I/O port 02H. The iUP system turns off a switched voltage supply line whenever an overcurrent condition is sensed on that line. Figure 10 contains switched voltage control and status bit definitions for the iUP system.

iPDS™ SWITCHED VOLTAGE PROGRAMMING

The iPDS system switches the +5.7 VSW, +VLOW, +VHIGH, and −12 VSW supply lines on and off under program control. The controlling program (either iPPS software or a user-written driver program) must write to I/O port E3H in order to turn on/off the required switched voltages. Figure 11 shows the bit definitions for programming the iPDS switched voltage lines.

User-Written iPDX™ Bus Drivers

User-written iPDX bus driver programs normally access plug-in modules designed for use with the iPDS system. A user-designed iPDX bus plug-in module can address a wide range of applications. The iPDX bus driver program for a user-designed plug-in module can range from simple (e.g., using a single I/O port to upload PROM data to the iPDS system), to complex (e.g., using nearly all the I/O ports to control a high-level instrumentation function).

The I/O ports available to the iPDX bus occupy addresses 10H through 1FH in the iPDS I/O space. Since both an I/O read and an I/O write are associated with each I/O address, the user has 32 I/O ports available for each driver program. Figure 12 is a blank chart that can be used to assign I/O addresses for a specific user driver program. Keep this chart for reference while writing the driver program.

The driver program must be written in 8085 code. Use no more than byte-wide transfers of address, data, and control information. The plug-in module can operate on information of virtually any bit length. The 8-bit width of the iPDX data bus imposes a byte-wide only requirement on all information transfers over the iPDX bus.

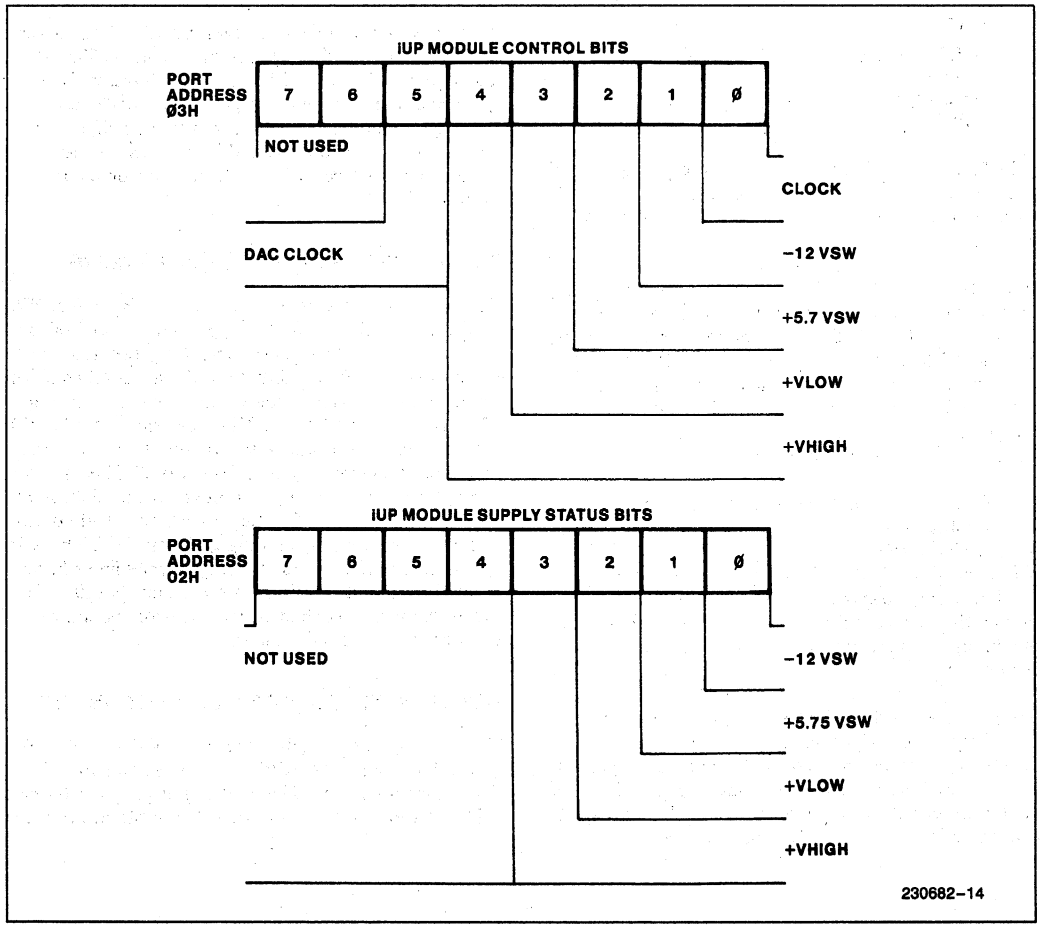


Figure 10. IUP Switched Voltage Control and Status Bit Definitions

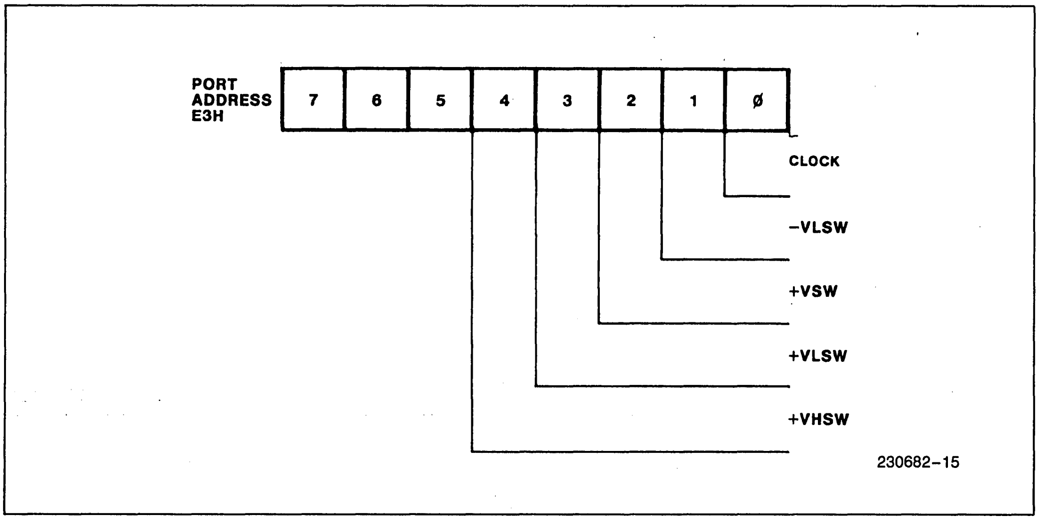


Figure 11. iPDS™ Switched Voltage Control Bit Definitions

I/O Port Address	I/O Write Active	I/O Read Active
10H		
11H		
12H		
13H		
14H		
15H		
16H		
17H		
18H		
19H		
1AH		
1BH		
1CH		
1DH		
1EH		
1FH		

Figure 12. Chart of iPDX Bus I/O Address Assignments

APPLICATION NOTE

AP-245

October 1986

Using Command Files to Speed Program Development

SRIVATS SAMPATH
DSO APPLICATIONS ENGINEERING

Order Number: 231481-001

INTRODUCTION

Recently, the computer industry has leaned toward providing a very friendly interface between human and machine:an interface that allows the user to be more productive in the least time possible, an interface that gives him or her the ability to use all of the computer's advanced features. Tools to assist the user include the command line interpreters (CLIs), HELP texts, and command file capability.

Recognizing the need for a sophisticated human interface, Intel has provided the advanced command line interpreter and syntax driver, HELP texts, and submit file capabilities on the Series IV Microcomputer Development System. This application note deals with the power and use of the Series IV command file capabilities. These facilities, available only on the Series IV and the network resource manager (NRM), represent a major improvement over the Series II and Series III systems.

Command files are files that can be executed by the host system. They are not programs. Typically, the computer recognizes the keystrokes from the keyboard and executes the operation selected. However, this becomes time-consuming for repetitive tasks involving many keystrokes. Constant user interaction is needed during the whole execution cycle, i.e., as soon as one operation is complete, the user has to type in the next command. If all these commands could be put in a file, and if the computer could read this file and execute all commands sequentially without any user interaction, we would have a system that drastically increases user productivity and reduces user fatigue.

Command files may be executed using two commands:SUBMIT and EXPORT. SUBMIT executes the command file instantaneously, while EXPORT sends the command file for execution at another system at a time determined by the DJC manager. (Refer to Application Note, AP-244, "DJC - The Key to Increased Network Productivity").

The SUBMIT command on the Series IV allows the user to replace commands typed in from the keyboard with ommands from a file. The submit command redirects console input to the specified file. The Series IV also has a utility editor called BATCH that helps the user in creating a submit file. BATCH is an editor that incorporates within it the Series IV syntax driver. The syntax driver is a human interface that keeps prompting the user for correct variables and options. With the BATCH utility, a user can create a submit file with no difficulty.

Command files can also be described as pseudoprogramming languages. These files can execute and perform logical operations, file I/O, support memory variables, looping, repeat function, and parameter passing. These commands execute their functions in a simple but effective way. They should be used with all other commands for more effectiveness. Command files resemble a very functional interpretive language. This is a very powerful utility on the Series IV and can be used to perform a variety of applications without requiring long programs to be written.

To illustrate these multiple features, this Application Note will describe a command file called MAILMAN, which is an internetwork mail utility. MAILMAN has the capability to mail across multiple networks using only existing software. The Series IV command file capability has been used extensively in this application.

COMMAND FILES AND THEIR CONSTRUCTS

This section describes all the command file operators, their functions and the interrelationships between them.

The LOG Command

One of the commands often used with a submit file is the LOG command. LOG copies all console output to the file specified.

For example, type this at the command line:

```
>LOG TEST.LOG           ;also send console output to TEST.LOG
>dir /APS1/USER.DIR expanded
iNDX-W31 (V2.8) DIR V2.8
DIRECTORY OF /APS1/JSER.DIR

FILE--NAME      OWNER--NAMEFILE--LENGTH  TYPE OWNER--ACCESS
JOHN.DIR        JOHN           2048      DIR  del dis add
CHRIS.DIR       CHRIS          4096      DIR  del dis add
SRIVAT.DIR      SRIVAT         2048      DIR  del dis add
NORI.DIR        NORI           2048      DIR  del dis add
WAYNE.DIR       WAYNE          4096      DIR  del dis add
WORLD.DIR       WORLD          2048      DIR  del dis add
SUPERUSER.DIR   BRIAN          2048      DIR  del dis add

total bytes used:   44021
>log :bb:                ;stop additional redirection to file
```

The LOG file will contain the exact output of the DIR command. We have selectively written specific data into a file that we will use later. This feature is very useful in command files and will be seen in MAILMAN.

In any system, the ability to pass parameters or variables from one program to another is very important. By passing parameters, a command file can be made to do a variety of tasks. The Series IV command file structure allows the passing of up to 10 parameters from the command line. These are designated %0 through %9.

For Example:

```
#CC86 %0.c debug %1

IF %status = 0 THEN

     link86 %0.obj &
     1/%1.lib,     &
     1/%2.lib,     &
     1/bvcslb.lib, &
     1/87null.lib, &
     to %0.86      &
bind          &
ss(stack(+800h),memory(+2800h))
END
```

The user locally invokes this command file by typing in:

```
>SUBMIT
COMPILE(CHECKEXIST,SMALL,SCLIB)
```

When the command file is executed, it substitutes:

CHECKEXIST	for %0
SMALL	for %1
SCLIB	for %2

The user can therefore, have one command file that can be used to compile and link different sources with different libraries just by specifying them at the command line level. This parameter-passing feature provides increased command file versatility. Without this feature, the user would have numerous command files, each executing a specific operation. This feature gives the user substantial flexibility and helps reduce the time to develop unique files for each application.

Command Line Variables

Command files also support the assignment of variables to alphanumericstrin gs through the SET command. SET assigns the value on the right to the variable name on the left.

For example:
```
SET NAME TO %0
```

will set the contents of %0 passed in from the command line to the CLI variable, NAME. Any further reference to %NAME will yield the value of %0. %NAME will access the contents of the variable, NAME. The Series IV CLI has an undocumented built in command called "DUMP". DUMP aids in debugging command files. It displays all the variables in a command file and their corresponding values.

For Example:

On submitting a file TEST.CSD

```
>SUBMIT TEST                will yield
>SET NAME1 to ""WAYNE''     ;set name1 to wayne
>SET NAME2 to ""SRIVAT''    ;set name2 to srivat
>SET NAME3 to ""JOHN''      ;set name3 to john
>DUMP                       ;show all variables and their values
NAME:""STATUS'' VALUE:""0''
NAME:""NAME1'' VALUE:""WAYNE''
NAME:""NAME2'' VALUE:""SRIVAT''
NAME:""NAME3'' VALUE:""JOHN''
EXIT COMMAND FILE /APS1/USER.DIR/SRIVAT.DIR/APNOTE.DIR/TES.CS
```

The system supports a predeclared variable called STATUS, which is set by the DQ$EXIT value of a previously executed program. For example, a successful termination will normally set STATUS to 0, while an error condition will return another value. For example, consider the UDI call DQ$EXIT(VALUE).

A dq$exit(0) will set STATUS to 0

A dq$exit(1) will set STATUS to 1

The value of STATUS depends on the parameter passed by the existing program. This tool is very useful for conditional compiles and conditional links. The command file can link and locate by monitoring STATUS only if the program is compiled without any errors.

Example:

```
cc86 %0.c debug
if %status = 0 then              ;link only if compile successful
     link86 %0.obj,         &
     l/sqmain.obj,          &
     l/sclib.lib,           &
     l/small.lib,           &
     l/bvcslb.lib,          &
     l/87null.lib           &
     to %0.86               &
     bind                   &
     ss(stack(+800h),memory(+2800h))
end
```

The command file above will link the object files only if the compiles are successful. This will save the time of whole like cycle without having the correct objects.

Most Intel-supplied software, especially the translators and utilities, use this concept. A successful program completion will exit with STATUS set to 0, and a program abort or termination will exit with STATUS set to something other than 0.

Accessing Data Files

The file I/O capabilities of command files are very powerful. The commands for file I/O are OPEN and READ. For example, consider the LOG file FILE.TMP, generated by the sequence:

```
LOG FILE.TMP
DIR /
LOG :BB:
```

FLE.TMP will contain:

```
>dir /
iNDX-W41 (V2.8) DIR V2.8
DIRECTORY OF /
FILE--NAMELOCATION  ACCESSIBILITY
APS--WO       remote
W1            remote
APSO          remote
APS1          remote
SYS             local
>log :bb:
```

A command file is shown that accesses this LOG file to discover the volume root name for the network:

```
OPEN file.tmp
COUNT 14
     read skip ;skip over the first 14 words of the file
     end
read root
end
```

In a DIR / command, the first file name is the volume root name. In this case, APS—W0 is the system volume root name and has to be assigned to some variable for future use. The command file utility OPEN for file I/O is used to gain access to the file FILE.TMP. The COUNT command is used as a loop counter that will loop around the number of times specified.

In this case, COUNT 14 will loop 14 times. Each time, it will set the variable SKIP from one word in the LOG file. A Word can be defined as a set of characters separated by a white space. We effectively skip over one word at a time. In this case, APS—W0 is the 14th word from the start of the file. So, the loop will skip 13 words and then read the system volume root name into the CLI variable ROOT. The COUNT, READ, OPEN and SKIP commands, built into the CLI, can be used only in submit files. The READ ROOT command will read the 14th word into the memory variable ROOT. %ROOT will contain the value of ROOT, which is, in this case, APS—W0.

Only one file can be opened at a time. There is no explicit CLOSE function. Opening another file will close the previous one. To force the Close of a parameter file, use the OPEN command on a nonexisting file. A combination of the LOG, OPEN, READ and SKIP commands help in doing very functional but effective file I/O

Conditional Command File Execution

Since the Series IV command file is like a pseudo-interpreter, it also supports logical operations such as IF, THEN, ELSE. The example below highlights how these constructs can be used within a command file.

This command file compiles any C program and then checks for a successful compile. If the compile is successful, the command file proceeds with linking and binding. If the compile is not successful, the file is an error reporter. More information on REPORT can be obtained from AP-244, an application note on distributed job control.

```
cc86 %0.c debug                          ;Compile the program
if %status 0 than                        ;If error in compile
    REPORT Error in compile of %0.c      ;Send message to user
else                                     ; and exit.
            REPORT Successful Compile. Proceeding with LINK
        link86 %0.obj,      &
        1/sqmain.obj,       &
        1/sclib.lib,        &
        1/small.lib,        &
        1/87null.lib        &
        to %0.86            &
        bind                &
        ss(stack(+800h),memory(+2800h))
if %status = 0 then                      ;Check for error in link
    REPORT Successful Link. End of Job.  ;If no error inform user
else
    REPORT Error while linking.....      ;If error inform userand
end                                      ;and exit.
```

Command File Looping

The COUNT construct is a looping control. REPEAT is an additional construct and works in conjunction with the WHILE and UNTIL commands. REPEAT will loop until the condition specified by the WHILE or UNTIL command is satisfied.

```
REPEAT
     UNTIL %STATUS = 2
          any operation
END
OR
REPEAT
     WHILE %STATUS <> 2
          any operation
END
```

These logical operators can be used in any combination as long as the syntax is correct. Each loop should have a matching end statement.

MAILMAN (A BRIEF EXPLANATION)

MAILMAN is a command file that allows users on one network to send mail to users on another network over an Ethernet cable. The network users do not have to distinguish or remember the USER/NRM configuration. MAIL is used as normal, and MAILMAN running as a background task knows the configuration and behaves accordingly. The MAILMAN utility uses existing software and the powerful constructs of the Series IV command file utility to illustrate that complex problems can be solved simply.

In a typical multiple network environment, communication between users on one network with users on the other network is very important. Since electronic mail supports only one network, there was a need for a system that supported mail over multiple networks. Writing a program in one of the high-level languages or assemblers using Ethernet protocols would require substantial time for designing, developing, and debugging.

The MAILMAN utility is an example of how command files increase productivity and help solve complex applications. MAILMAN, which uses almost all the commands and constructs supported by command files, will help the reader understand how command files can be used for any particular application.

In this example two NRMs will send mail between each other by executing the MAILMAN utility on import stations at both ends. These import stations import from a utility queue called iNDXUTILITY.Q. For more information on queues and remote job execution, refer to Application Note AP-244 titled DJC:Key to Increased Network Productivity.

MAILMAN generates several data files using the LOG command during execution to discover the various system variables. It also depends on two data files called REMOTE.USERS and LOCAL.USERS. These files have the sameformat and are used to distinguish which NETWORK each user is Sysgenned onto. The format is:

```
Line 1:NRM root volume name
Line 2:Username
Line 3:Username
•
•
•
Last line:blank <to signify the end of
list>
```

These data files should be placed in the directory MAIL.DIR of each system.

For example, the file LOCAL.USERS under /APS—w0 will look like:

```
APS--WO
SRIVAT
WAYNE
JOHN
CHRIS
<blank>
```

And the file REMOTE.USERS will look like:

```
PMO
PAUL
FRANCIS
STU
SUNIL
<blank>
```

Each of the REMOTE users have to be sysgenned onto the local network as users but without a home directory. The user MAILMAN has to be sysgenned intothe network as a user with a home directory.

MAILMAN (THE COMMAND FILE)

```
;*****************************************************************************
;* CHECKMAIL.CSD . This is a submit file that allows multiple NRM mail.      *
;* A detailed explanation of the system requirements is in the CHECKMAIL.DOC*
;* file. CHECKMAIL allows users on one NRM to mail messages to users on      *
;* other NRMS. There is no limit to the number of NRM's, but you must read   *
;* the Toolbox manual or CHECKMAIL.DOC to effect modifications for more than*
;* two NRMs.  This file is set up for two NRMs only.                         *
;*****************************************************************************
;*****************************************************************************
; Need to know the root volume name
;*****************************************************************************
log file.tmp
dir /
log :bb:
open file.tmp
count 14
    read skip
    end
read root
open file2.tmp                          ; To close param.file
;*****************************************************************************
; If the user did not supply a parameter use their USER NAME
;*****************************************************************************
if %Ø <> ""
    set name to %Ø
else
    log file.tmp ; Who is currently using this command file
    id
    log :bb:
    open file.tmp
    read skip,skip,skip,skip,name
    end
;*****************************************************************************
; If the user is MAILMAN then we are in receive mode
; else we are in transmit mode
;*****************************************************************************
if %name <> MAILMAN then
; Can now check for mail
    delete message.found
    mail box %name
    save 1 message.found
    q
; Check to see if there was a message in the mailbox
    checkexist message.found
    if %status = 1
        set sent to false
        open /%root/mail.dir/remote.users
; Read the root volume of the remote network
        read rroot
        repeat
            while %sent = false
```

231481-1

```
; Read one of the remote system user names
            read remote
            while %remote <> ""
            if %name = %remote then
                open file2.tmp
                log file.tmp
                time
                log :bb:
                open file.tmp
                count 11
                    read skip
                    end
                read time
                nncopy message.found to %rroot/mail.dir/%name/%time    &
                    username (mailman ) password (post) nrm (Ø)
;***************************************************************************
; The message has been forwarded to the other system,
;     remove it from the local mailbox
;***************************************************************************
                mail box %name
                delete 1
                e
                set sent to true
                end
            end
        if %sent = false then
; This is a local user message
            Report You have mail in box %name
            end
        end
;***************************************************************************
; The user MAILMAN, operate in receive mode
;***************************************************************************
else
    set name to user
    repeat
    open /%root/mail.dir/local.users
; Ignore the root name
    read skip
; Skip to the user name
    read %name
    while %user <> ""
    log file.tmp
;***************************************************************************
; Check for 'timed' mail delivery from another system
;***************************************************************************
    dir /%root/mail.dir/%user for ??:??:??
    log :BB:
    open file.tmp
    count 16
        read skip
        end
    repeat
        read file
        while %file <> ""
```

231481-2

```
107 ;*************************************************************************
108 ; A 'timed' message has been found, MAIL it to the appropriate user
109 ;*************************************************************************
110         mail /%root/mail.dir/%user/%file to %user &
111         subject(Arrived at %FILE and forwarded by Mailman)
112         delete /%root/mail.dir/%user/%file
113         end
114     set name to "skip,%name"
115     end
116 end
117 ;*************************************************************************
118 ; All done, reinvoke myself
119 ;*************************************************************************
120 export /%root/checkmail(%0) to iNDXutility.q nolog
```

231481-3

MAILMAN (AN IN-DEPTH LOOK)

Many comments, included to help your understanding of the program flow, could be edited out to speed execution. There are some concepts used in this program that need additional explanation.

#65

We use the NRM to NRM communications package discussed in Application Note AP-241 (Multiple NRM Ethernet Communications) to communicate between the NRMS. If other NRMs were not on the same Ethernet cable, we could change this line only to incorporate autodialing to other NRMs.

#114

"SET NAME TO "SKIP", %NAME"

As described earlier, the CLI can have only one file open at any time. The opening of any other file will close the previously opened file. In line #88, we open the file LOCAL.USERS and skip to the first name. Once this user name has been established, we need to check for mail in his or her Mail directory. This operation is done in lines #99 to #114. However, we now need the name of the next user. Since we already closed the file LOCAL.USERS, the next time lines #83 through #94 are executed, the file pointer will point to the same name and will repeat the loop. By setting NAME to "SKIP, %name", the next time the file is opened, it will automatically read the second name. The CLI variable %NAME will be "SKIP, %NAME" and the command READ will skip one word and read the next. The third time, NAME will be "SKIP,SKIP, %NAME", the fourth time, "SKIP, SKIP, SKIP, %NAME". Even though the file LOCAL.USERS is constantly opened and closed, our file pointer is still intact and points to the correct word.

#120

"Export /%root/CHECKMAIL(%0) to indxutility.q nolog"

We need this file to be executing forever:checking the mail and sending it to the right networks. This is an example of looping in export files, i.e., having a remote job run forever. For more information, refer to the Application Note AP-244 DJC, Key to Increased Network Productivity.

#44

CHECKEXIST.86 is a file checker. If the specified file exists, it will exit with STATUS set to 1. Otherwise, STATUS is set to 0. This is used to determine the existence of a file. This small utility was developed for use within the MAILMAN package.

Example:
```
CHECKEXIST TEST.FILE will
set STATUS to 1 if TEST.FILE exists
else
set STATUS to 0
```

Within the MAILMAN utility, CHECKEXIST is used to check if any mail messages exist for the user specified. Looking at the source of CHECKEXIST.C shown in Appendix A, the concept of STATUS becomes very clear.

CONCLUSION

MAILMAN is an extensive example of the power of the Series IV command file capabilities. The complete file was developed and debugged in less than a week - far shorter than writing an application program to talk over Ethernet. The Series IV command file capability enables you to build upon software you already have to reach higher heights more quickly.

APPENDIX A (CHECKEXIST.86)

```
/**************************************************************************/
/*  CHECKEXIST.C          Existence checker for files.                   */
/*  CHECKEXIST will exit with an exit code of "Ø" if file is not found */
/*  else it will return with an exit code of 1. This will be passed     */
/*  into %STATUS in a command file.                                       */
/*  Syntax for CHECKEXIST.86 :                                            */
/*  CHECKEXIST < filename >                                               */
/*                                                                        */
/*  Example:                                                              */
/*                                                                        */
/*  The batch file CHECK.CSD contains these statements.                   */
/*  CHECKEXIST My.File                                                    */
/*                                                                        */
/*  Submitting this batch file will set %STATUS to Ø if file does not   */
/*  exist and 1 if file exists.                                           */
/*                                                                        */
/*  For an example of the use of this CUSP see the Mailman example for */
/*  multiple network mail.                                                */
/**************************************************************************/

#include <:f2:stdio.h>
#include <:f2:ctype.h>

/**************************************************************************/
/*   Start of main routine                                                */
/**************************************************************************/
main(argc, argv)

int        argc;
char    *argv[];

{

    FILE *fp, *fopen();

    if (argc == 1)
    {
        puts("Error..Filename not specified.");
    }
    /* Exit with return value Ø if file does not exist */
    /* else exit with a 1                              */
    if ((fp = fopen(*++argv, "r")) == NULL)
    {
        dq$exit(Ø);
    }
    else
    {
        fclose(fp);
        printf("FOUND FILE : %s\n",*argv);
        dq$exit(1);
    }

}
```

231481-4

System Design Kits

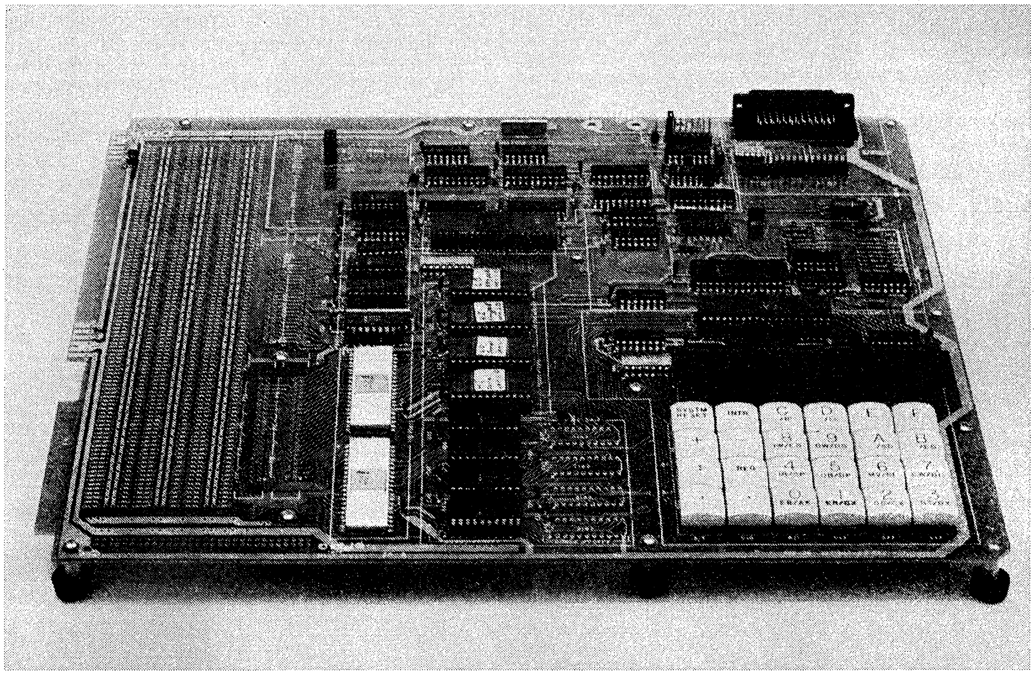
6

intel

SDK-86
MCS®-86 SYSTEM DESIGN KIT

- **Complete Single Board Microcomputer System Including CPU, Memory, and I/O**
- **Easy to Assemble Kit Form**
- **High Performance 8086 16-Bit CPU**
- **Interfaces Directly with TTY or CRT**
- **Interactive LED Display and Keyboard**
- **Wire Wrap Area for Custom Interfaces**
- **Extensive System Monitor Software in ROM**
- **Comprehensive Design Library Included**

The SDK-86 MCS-86 System Design Kit is a complete single board 8086 microcomputer system in kit form. It contains all necessary components to complete construction of the kit, including LED display, keyboard, resistors, caps, crystal, and miscellaneous hardware. Included are preprogrammed ROMs containing a system monitor for general software utilities and system diagnostics. The complete kit includes an 8-digit LED display and a mnemonic 24-key keyboard for direct insertion, examination, and execution of a user's program. In addition, it can be directly interfaced with a teletype terminal, CRT terminal, or the serial port of an Intellec system. The SDK-86 is a high performance prototype system with designed-in flexibility for simple interface to the user's application.



205945–1

October 1986
Order Number: 205945-002

FUNCTIONAL DESCRIPTION

The SDK-86 is a complete MCS-86 microcomputer system on a single board, in kit form. It contains all necessary components to build a useful, functional system. Such items as resistors, caps, and sockets are included. Assembly time varies from 4 to 10 hours, depending on the skill of the user. The SDK-86 functional block diagram is shown in Figure 1.

8086 Processor

The SDK-86 is designed around Intel's 8086 microprocessor. The Intel 8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CERDIP package. The processor features attributes of both 8-bit and 16-bit microprocessors in that it addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance. Additional features of the 8086 include the following:

- Direct addressing capability to one megabyte of memory
- Assembly language compatibility with 8080/8085
- 14 word x 16-bit register set with symmetrical operations
- 24 operand addressing modes
- Bit, byte, word, and block operations
- 8 and 16-byte signed and unsigned arithmetic in binary or decimal mode, including multiply and divide
- 4 or 5 or 8 MHz clock rate

A block diagram of the 8086 microprocessor is shown in Figure 2.

System Monitor

A compact but powerful system monitor is supplied with the SDK-86 to provide general software utilities and system diagnostics. It comes in preprogrammed read only memories (ROMs).

Communications Interface

The SDK-86 communicates with the outside world through either the on-board light emitting diode (LED) display/keyboard combination or the user's TTY or CRT terminal (jumper selectable), or by means of a special mode in which an Intellec development system transports finished programs to and from the SDK-86. Memory may be easily expanded by simply soldering in additional devices in locations provided for this purpose. A large area of the board (22 square inches) is laid out as general purpose wire-wrap for the user's custom interfaces.

Assembly

Only a few simple tools are required for assembly: soldering iron, cutters, screwdriver, etc. The SDK-86 assembly manual contains step-by-step instructions for easy assembly with a minimum of mistakes. Once construction is complete, the user connects his kit to a power supply and the SDK-86 is ready to go. The monitor starts immediately upon power-on or reset.

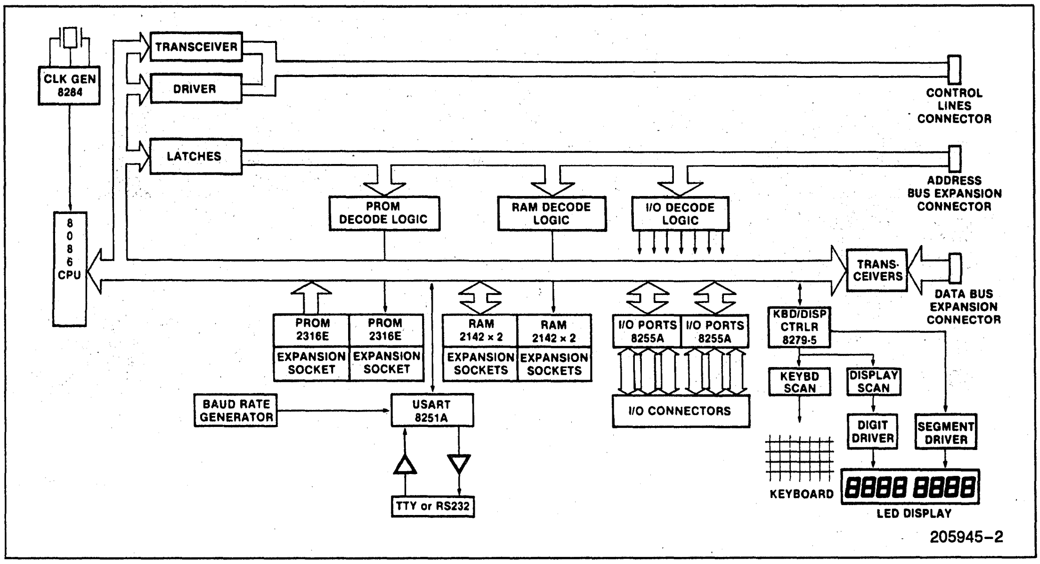


Figure 1. SDK-86 System Design Kit Functional Block Diagram

Commands—Keyboard mode commands, serial port commands, and Intellec slave mode commands are summarized in Table 1, Table 2, and Table 3, respectively. The SDK-86 keyboard is shown in Figure 3.

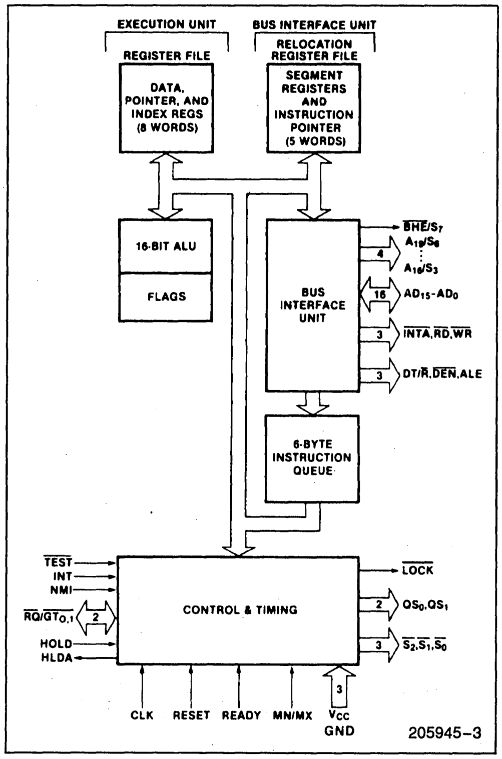


Figure 2. 8086 Microprocessor Block Diagram

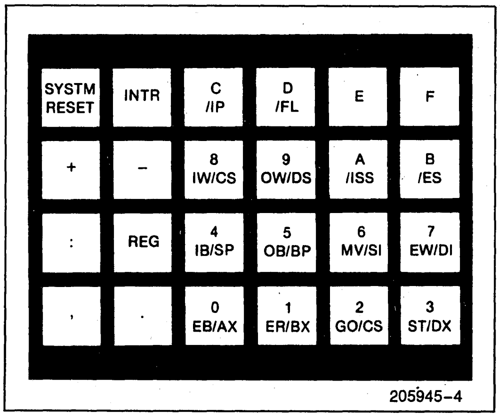


Figure 3. SDK-86 Keyboard

Documentation

In addition to detailed information on using the monitors, the SDK-86 user's manual provides circuit diagrams, a monitor listing, and a description of how the system works. The complete design library for the SDK-86 is shown in Figure 4 and listed in the specifications section under Reference Manuals.

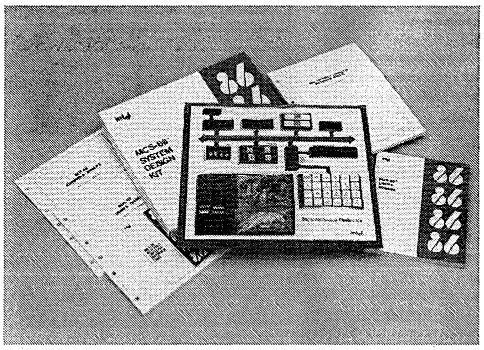


Figure 4. SDK-86 Design Library

Table 1. Keyboard Mode Commands

Command	Operation
Reset	Starts monitor.
Go	Allows user to execute user program, and causes it to halt at predetermined program stop. Useful for debugging.
Single Step	Allows user to execute user program one instruction at a time. Useful for debugging.
Substitute Memory	Allows user to examine and modify memory locations in byte or word mode.
Examine Register	Allows user to examine and modify 8086 register contents.
Block Move	Allows user to relocate program and data portions in memory.
Input or Output	Allows direct control of SDK-86 I/O facilities in byte or mode.

Table 2. Serial Mode Commands

Command	Operation
Dump Memory	Allows user to print or display large blocks of memory information in hex format than amount visible on terminal's CRT display.
Start/Continue Display	Allows user to display blocks of memory information larger than amount visible on terminal's CRT display.
Punch/Read Paper Tape	Allows user to transmit finished programs into and out of SDK-86 via TTY paper tape punch.

8086 INSTRUCTION SET

Table 3 contains a summary of processor instructions used for the 8086 microcprocessor.

Table 3. 8086 Instruction Set Summary

Mnemonic and Description	Instruction Code			
Data Transfer				
MOV = Move:	76543210	76543210	76543210	76543210
Register/Memory to/from Register	100010dw	mod reg r/m		
Immediate to Register/Memory	1100011w	mod 000 r/m	data	data if w = 1
Immediate to Register	1011wreg	data	data if w = 1	
Memory to Accumulator	1010000w	addr-low	addr-high	
Accumulator to Memory	1010001w	addr-low	addr-high	
Register/Memory to Segment Register	10001110	mod 0 reg r/m		
Segment Register to Register/Memory	10001100	mod 0 reg r/m		
PUSH = Push:				
Register/Memory	11111111	mod 110 r/m		
Register	01010reg			
Segment Register	000reg110			
POP = Pop:				
Register/Memory	10001111	mod 000 r/m		
Register	01011reg			
Segment Register	000reg111			
XCHG = Exchange:				
Register/Memory with Register	1000011w	mod reg r/m		
Register with Accumulator	10010reg			
IN = Input				
Fixed Port	1110010w	port		
Variable Port	1110110w			
OUT = Output				
Fixed Port	1110011w	port		
Variable Port	1110111w			
XLAT = Translate Byte to AL	11010111			
LEA = Load EA to Register	10001101	mod reg r/m		
LDS = Load Pointer to DS	11000101	mod reg r/m		
LES = Load Pointer to ES	11000100	mod reg r/m		
LAHF = Load AH with Flags	10011111			
SAHF = Store AH into Flags	10011110			
PUSHF = Push Flags	10011100			
POPF = Pop Flags	10011101			
Arithmetic				
ADD = Add:				
Reg./Memory with Register to Either	000000dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 000 r/m	data	data if s w = 01
Immediate to Accumulator	0000010w	data	data if w = 1	
ADC = Add with Carry:				
Reg./Memory with Register to Either	000100dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 010 r/m	data	data if s w = 01
Immediate to Accumulator	0001010w	data	data if w = 1	
INC = Increment:				
Register/Memory	1111111w	mod 000 r/m		
Register	01000reg			
AAA = ASCII Adjust for Add	00110111			
BAA = Decimal Adjust for Add	00100111			
SUB = Subtract:				
Reg./Memory and Register to Either	001010dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 101 r/m	data	data if s w = 01
Immediate from Accumulator	0010110w	data	data if w = 1	
SSB = Subtract with Borrow				
Reg./Memory and Register to Either	000110dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 011 r/m	data	data if s w = 01
Immediate from Accumulator	000111w	data	data if w = 1	
DEC = Decrement:				
Register/memory	1111111w	mod 001 r/m		
Register	01001 reg			
NEB = Change sign	1111011w	mod 011 r/m		

Mnemonic and Description	Instruction Code			
CMP = Compare:	76543210	76543210	76543210	76543210
Register/Memory and Register	001110dw	mod reg r/m		
Immediate with Register/Memory	100000sw	mod 111 r/m	data	data if s w = 01
Immediate with Accumulator	0011110w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	00111111			
DAS = Decimal Adjust for Subtract	00101111			
MUL = Multiply (Unsigned)	1111011w	mod 100 r/m		
IMUL = Integer Multiply (Signed)	1111011w	mod 101 r/m		
AAM = ASCII Adjust for Multiply	11010100	00001010		
DIV = Divide (Unsigned)	1111011w	mod 110 r/m		
IDIV = Integer Divide (Signed)	1111011w	mod 111 r/m		
AAD = ASCII Adjust for Divide	11010101	00001010		
CBW = Convert Byte to Word	10011000			
CWD = Convert Word to Double Word	10011001			
Logic				
NOT = Invert	1111011w	mod 010 r/m		
SHL/SAL = Shift Logical/Arithemtic Left	110100vw	mod 100 r/m		
SHR = Shift Logical Right	110100vw	mod 101 r/m		
SAR = Shift Arithmetic Right	110100vw	mod 111 r/m		
ROL = Rotate Left	110100vw	mod 000 r/m		
ROR = Rotate Right	110100vw	mod 001 r/m		
RCL = Rotate Through Carry Flag Left	110100vw	mod 010 r/m		
RCR = Rotate Through Carry Right	110100vw	mod 011 r/m		
AND = And:				
Reg./Memory and Register to Either	001000dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 100 r/m	data	data if w = 1
Immediate to Accumulator	0010010w	data	data if w = 1	
TEST = And Function to Flags. No Result:				
Register/Memory and Register	1000010w	mod reg r/m		
Immediate Data and Register/Memory	1111011w	mod 000 r/m	data	data if w = 1
Immediate Data and Accumulator	1010100w	data	data if w = 1	
OR = Or:				
Reg./Memory and Register to Either	000010dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 001 r/m	data	data if w = 1
Immediate to Accumulator	0000110w	data	data if w = 1	
XOR = Exclusive or:				
Reg./Memory and Register to Either	001100dw	mod reg r/m		
Immediate to Register/Memory	1000000w	mod 110 r/m	data	data if w = 1
Immediate to Accumulator	0011010w	data	data if w = 1	
String Manipulation				
REP = Repeat	1111001z			
MOVS = Move Byte/Word	1010010w			
CMPS = Compare Byte/Word	1010011w			
SCAS = Scan Byte/Word	1010111w			
LODS = Load Byte/Wd to AL/AX	1010110w			
STOS = Stor Byte/Wd from AL/A	1010101w			
Control Transfer				
CALL = Call:				
Direct Within Segment	11101000	disp-low	disp-high	
Indirect Within Segment	11111111	mod 010 r/m		
Direct Intersegment	10011010	offset-low	offset-high	
		seg-low	seg-high	
Indirect Intersegment	11111111	mod 011 r/m		

Table 4. 8086 Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code		
	76543210	76543210	76543210
JMP = Unconditional Jump:			
Direct Within Segment	11101001	disp-low	disp-high
Direct Within Segment-Short	11101011	disp	
Indirect Within Segment	11111111	mod 1 0 0 r/m	
Direct Intersegment	11101010	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	11111111	mod 1 0 1 r/m	
RET = Return from CALL:			
Within Segment	11000011		
Within Seg Adding Immed to SP	11000010	data-low	data-high
Intersegment	11001011		
Intersegment Adding Immediate to SP	11001010	data-low	data-high
JE/JZ = Jump on Equal/Zero	01110100	disp	
JL/JNGE = Jump on Less/Not Greater or Equal	01111100	disp	
JLE/JNG = Jump on Less/Not Greater	01111110	disp	
JB/JNAE = Jump on Below/Not Above or Equal	01110010	disp	
JBE/JNA = Jump on Below or Equal/ Not Above	01110110	disp	
JP/JPE = Jump on Parity/Parity Even	01111010	disp	
JO = Jump on Overflow	01110000	disp	
JS = Jump on Sign	01111000	disp	
JNE/JNZ = Jump on Not Equal/Not Zero	01110101	disp	
JNL/JGE = Jump on Not Less/Greater or Equal	01111101	disp	
JNLE/JG = Jump on Not Less or Equal/ Greater	01111111	disp	
JNG/JAE = Jump on Not Below/Above or Equal	01110011	disp	
JNBE/JA = Jump on Not Below or Equal/Above	01110111	disp	
JNP/JPO = Jump on Not Par/Par Odd	01111011	disp	
JNO = Jump on Not Overflow	01110001	disp	
JNS = Jump on Not Sign	01111001	disp	

Mnemonic and Description	Instruction Code	
	76543210	76543210
LOOP = Loop CX Times	11100010	disp
LOOPZ/LOOPE = Loop While Zero/Equal	11100001	disp
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	11100000	disp
JCXZ = Jump on CX Zero	11100011	disp
INT = Interrupt		
Type Specified	11001101	Type
Type 3	11001100	
INTO = Interrupt on Overflow	11001110	
IRET = Interrupt Return	11001111	
Processor Control		
CLC = Clear Carry	11111000	
CMC = Complement Carry	11110101	
STC = Set Carry	11111001	
CLD = Clear Direction	11111100	
STD = Set Direction	11111101	
CLI = Clear Interrupt	11111010	
STI = Set Interrupt	11111011	
HLT = Halt	11110100	
WAIT = Wait	10011011	
ESC = Escape (to External Device)	11011xxx	mod x x x r/m
LOCK = Bus Lock Prefix	11110000	

NOTES:
AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive:
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction
if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BX) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)
*except if mod = 00 and r/m = then EA = disp-high: disp-low.

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 0 then "count" in (CL)
x = don't care
if v = 0 then "count" = 1; if v = 1 then "count" in (CL) register
z is used for string primitives for comparison with ZF FLAG
SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:
FLAGS =
X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

SPECIFICATIONS

Central Processor

CPU—8086 (5 MHz clock rate)

NOTE:
May be operated at 2.5 MHz or 5 MHz, jumper selectable, for use with 8086.

Memory

ROM: 8K bytes 2316/2716

RAM: 2K bytes (expandable to 4K bytes) 2142

Addressing

ROM: FE000-FFFFF

RAM: 0–7FF (800-FFF available with additional 2142's)

NOTE:
The wire-wrap area of the SDK-86 PC board may be used for additional custom memory expansion.

Input/Output

Parallel: 48 lines (two 8255A's)

Serial: RS232 or current loop (8251A)

Baud Rate: selectable from 110 to 4800 baud

Interfaces

Bus: All signals TTL compatible

Parallel I/O: All signals TTL compatible

Serial I/O: 20 mA current loop TTY or RS232

NOTE:
The user has access to all bus signals which enable him to design custom system expansions into the kit's wire-wrap area.

Interrupts (256 vectored)

Maskable

Non-maskable

TRAP

DMA

Hold Request: Jumper selectable. TTL compatible input.

Software

System Monitor: Preprogrammed 2716 or 2316 ROMs

Addresses: FE000–FFFFF

Monitor I/O: Keyboard/display or TTY or CRT (serial I/O)

Physical Characteristics

Width: 13.5 in. (34.3 cm)
Height: 12 in. (30.5 cm)
Depth: 1.75 in. (4.45 cm)
Weight: approx. 24 oz. (3.3 kg)

Electrical Characteristics

DC Power Requirement
(Power supply not included in kit)

Voltage	Current
V_{CC} 5V ±5%	3.5A
V_{TTY} − 12V ±10%	0.3A (V_{TTY} required only if teletype is connected)

Environmental Characteristics

Operating Temperature: 0°C– +50°C

Reference Manuals

9800697A—SDK-86 MCS-86 System Design Kit Assembly Manual

9800722—MCS-86 User's Manual

9800640A—8086 Assembly Language Programming Manual

8086 Assembly Language Reference card

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

Part Number	Description
SDK-86	MCS-86 System Design Kit

PROM Programming 7

iUP-200A/iUP-201A UNIVERSAL PROM PROGRAMMERS

MAJOR iUP-200A/iUP-201A FEATURES:

- **Personality Module Plug-Ins Provide Industry First Support for Intel and Intel Compatible EPROMs, EEPROMs, KEPROM, Microcontrollers, and other Programmable Devices.**
- **Powerful PROM Programming Software (iPPS) Makes Programming Easy with Intellec® Development System, iNDS-II Networks, iPDS Personal Development System, IBM P/C, X/T, A/T, and PC DOS Compatibles.**
- **New Modules Provide Industry-Fastest, Intelligent Programming Algorithms to Dramatically Shorten Programming Times.**
- **iUP-200A Provides On-Line Operation with a Built-In Serial RS232 Interface and Software for a Growing List of Environments.**
- **iUP-201A Provides Same On-Line Performance and Adds Keyboard and Display for Stand-Alone Use.**
- **iUP-201A Stand-Alone Capability Includes Device Previewing, Editing, Duplication, and Download from any Source Over RS232C Port.**
- **Regular Updates and Add-Ons Have Maintained Even the Earliest iUP-200 and iUP-201 Users at the State-of-Art.**

The iUP-200A and iUP-201A universal programmers program and verify data in all the Intel and Intel compatible, programmable devices (EPROMs and EEPROMs). They can also program the EPROM memory portions of Intel's single-chip microcomputer and peripheral devices. The iUP-200A and iUP-201A universal programmers provide on-line programming and verification in a growing variety of development environments using the Intel PROM programming software (iPPS). In addition, the iUP-201A universal programmer supports off-line, stand-alone program editing, EPROM duplication, and EPROM memory locking. The iUP-200A universal programmer is expandable to an iUP-201A model.

210319–1

October 1986
Order Number: 210319-004

FUNCTIONAL DESCRIPTION

The iUP-200A universal programmer operates in on-line mode. The iUP-201A universal programmer operates in both on-line and off-line mode.

On-Line System Hardware

The iUP-200A and iUP-201A universal programmers are free-standing units that, when connected to a host computer with at least 64K bytes of memory, provide on-line EPROM programming and verification of Intel programmable devices. In addition, the universal programmer can read the contents of the ROM versions of these devices.

The universal programmer communicates with the host through a standard RS232C serial data link. Different versions of the iUP-200A and iUP-201A are equipped with different cables, including the cable most commonly used for interfacing to that host. Care should be taken that the version with the correct cable for your particular system is selected, as cable requirements can vary with your host configuration. A serial converter is needed when using the MDS 800 as a host system. (Serial converters are available from other manufacturers).

Each universal programmer contains the CPU, selectable power supply, static RAM, programmable timer, interface for personality modules, RS232C interface for the host system, and control firmware in EPROM. The iUP-201A also has a keyboard and display.

A personality module adapts the universal programmer to a family of EPROM devices; it contains all the hardware and firmware necessary to program either a family of Intel EPROMs or a single Intel device. The user inserts the personality module into the universal programmer front panel.

Figure 1 shows the iUP-200A on-line system configurations, and Figure 2 shows the on-line system data flow.

On-Line System Software

The iUP-200A and iUP201A includes your choice of one copy of Intel's PROM Programming software iPPS, selected from a growing list of versions for different operating systems and hosts. Each version includes the software implementation designed for that host and O.S. and the RS232C cable most commonly used. Additional versions may be purchased separately if you decide to change hosts at a later date. The iPPS software provides user control through an easy-to-use interactive interface. The iPPS software performs the following functions to make EPROM programming quick and easy:

- Reads EPROMs and ROMs
- Programs EPROMs directly or from a file
- Verifies EPROM data with buffer data
- Locks EPROM memory from unauthorized access (on devices which support this feature)
- Prints EPROM contents on the network or development system printer
- Performs interactive formatting operations such as interleaving, nibble swapping, bit reversal, and block moves
- Programs multiple EPROMs from the source file, prompting the user to insert new EPROMs
- Uses a buffer to change EPROM contents

All iPPS commands, as well as program address and data information, are entered through the host system ASCII keyboard and displayed on the system CRT. Table 1 summarizes the iPPS commands.

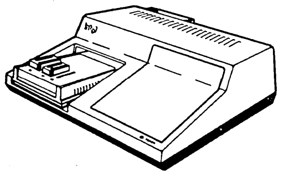
The iPPS software lets the user load programs into an EPROM from host system memory or directly from a disk file. Access to the disk lets the user create and manipulate data in a virtual buffer with an address range up to 16M. This large block of data can be formatted into different EPROM word sizes for program storage into several different EPROM types. In addition, a program stored in the target EPROM, the host system memory, or a system disk file can be interleaved with a second program and entered into a specific target EPROM or EPROMs.

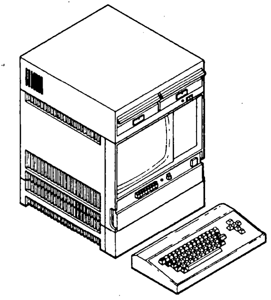
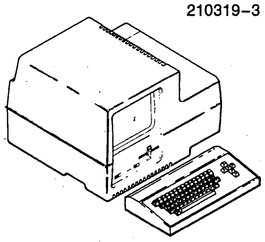
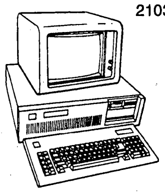
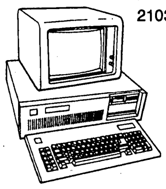
The iPPS software supports data manipulation in any Intel format: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, and 80286 absolute object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. The user can easily change default data formats as well as number bases.

The user invokes the iPPS software from the operating system. Intellec and iPDS Development Systems running ISIS allow running the software under control of ISIS submit files freeing the operator from repetitious command entry.

System Expansion

The iUP-200A universal programmer can be easily upgraded (by the user) to an iUP-201A universal programmer for off-line operation. The upgrade kit (iUP-PAK-A) is available from Intel or your local Intel distributor.


210319-2

MODEL NUMBER	HOST SUPPORTED	S/W OPERATING SYSTEM ENVIRONMENT	RS232C CABLE INCLUDED*	I/O PORT USED
iUP200A211A OR iUP200A212B	 210319-3	ISIS II	INTELLEC-STYLE	CH1 OR 2
iUP200A213C	 210319-4	iNDX	INTELLEC-STYLE	CH1 OR 2
iUP200A216D	 210319-5	PC-DOS	PC OR XT STYLE	COM 1 OR 2
iUP200A217D	 210319-6	PC-DOC	PC AT STYLE	COM 1 OR 2

*RS232C CABLES: INTELLEC STYLE—DB 25 PIN MALE TO DB 25 PIN MALE, NULL MODEM CABLE.

PC XT STYLE — DB 25 PIN FEMALE TO DB 25 PIN MALE, NULL MODEM CABLE.

PC AT STYLE — DB 9 PIN FEMALE TO DB 25 PIN MALE CABLE.

Figure 1. On-Line System Configurations

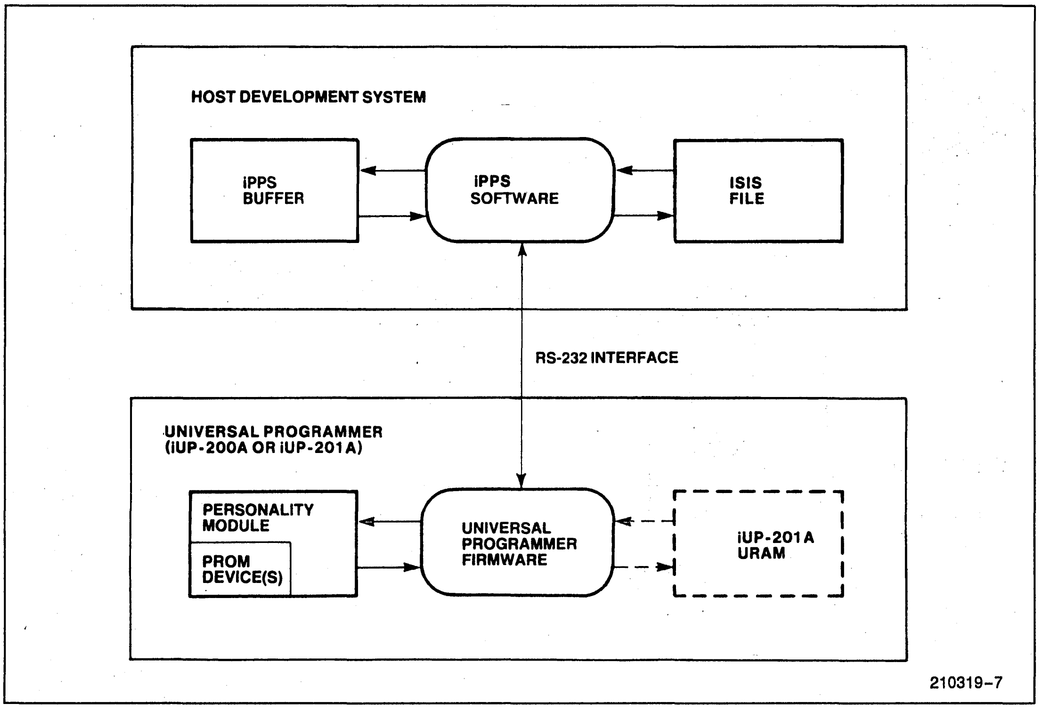


Figure 2. On-Line System Data Flow

Off-Line System

The iUP-201A universal programmer has all the on-line features of the iUP-200A universal programmer plus off-line editing, EPROM duplication, program verification, and locking of EPROM memory independent of the host system. The iUP-201A universal programmer also accepts Intel hexadecimal programs developed on non-Intel development systems. Just a few keystrokes download the program into the iUP RAM for editing and loading into a EPROM.

Off-line commands are entered using the off-line command keys summarized in Table 2.

In addition to the hardware components included as part of the iUP-200A, the iUP-201A contains a 24-character alphanumeric display, full hexadecimal 12-function keypad, and 32K bytes of iUP RAM. Figure 3 illustrates the iUP-201A keyboard and display.

The two logical devices accessible during off-line operation are the EPROM device and the iUP RAM. A typical operation is copying the data from an EPROM (or ROM) into the iUP RAM, modifying this data in iUP RAM, and programming the modified data back into a EPROM device. The address range of the iUP RAM is automatically determined by the universal programmer when EPROM type selection is made. Figure 4 shows the off-line system data flow.

SYSTEM DIAGNOSTICS

Both the iUP-200A and iUP-201A universal programmers include self-contained system diagnostics that verify system operation and aid the user in fault isolation. Diagnostics are performed on the power supply, CPU internal firmware ROM, internal RAM, timer, the iUP-201A keyboard, and the iUP RAM. In addition, tests are made on any personality module installed in the programmer the first time the module is accessed. The personality module tests include the power select circuitry and module firmware. Straightforward messages are provided on the development system display in on-line mode and on the iUP-201A display in off-line mode.

PERSONALITY MODULES

A personality module is the interface between the iUP-200A/iUP-201A universal programmer (or an iPDS system) and a selected EPROM (or ROM). Personality modules contain all the hardware and

Table 1. iPPS Command Summary

Command	Description
PROGRAM CONTROL GROUP	CONTROLS EXECUTION OF THE iPPS SOFTWARE.
EXIT	Exits the iPPS software and returns control to the ISIS operating system.
<ESC>	Terminates the current command.
REPEAT	Repeats the previous command.
ALTER	Edits and re-executes the previous command.
UTILITY GROUP	DISPLAYS USER INFORMATION AND STATUS AND SETS DEFAULT VALUES.
DISPLAY	Displays EPROM, buffer, or file data on the console.
PRINT	Prints EPROM, buffer, or file data on the local printer.
QUEUE	Prints EPROM, buffer or file data on the network spooled printer.
HELP	Displays user assistance information.
MAP	Displays buffer structure and status.
BLANKCHECK	Checks for unprogrammed EPROMs.
OVERLAY	Checks whether non-blank EPROMs can be programmed.
TYPE	Selects the EPROM type.
INITIALIZE	Initializes the default number base and file type.
WORKFILES	Specifies the drive device for temporary work files.
BUFFER GROUP	EDITS, MODIFIES, AND VERIFIES DATA IN THE BUFFER.
SUBSTITUTE	Examines and modifies buffer data.
LOADDATA	Loads a section of the buffer with a constant.
VERIFY	Verifies data in the EPROM with buffer data.
FORMATTING GROUP	REARRANGES DATA FROM THE EPROM, BUFFER, OR FILE.
FORMAT	Formats and interleaves buffer, EPROM, or file data.
COPY GROUP	COPIES DATA FROM ONE DEVICE TO ANOTHER.
COPY (file to PROM)	Programs the EPROM with data in a file on disk.
COPY (PROM to file)	Saves EPROM data in a file on disk.
COPY (buffer to PROM)	Programs the EPROM with data from the buffer.
COPY (PROM to buffer)	Loads the buffer with data in the EPROM.
COPY (buffer to file)	Saves the contents of the buffer in a file on disk.
COPY (file to buffer)	Loads the buffer from a file on disk.
COPY (file to URAM)	Loads file data into the iUP RAM (iUP-201A model only).
COPY (URAM to file)	Saves iUP URAM data in a file on disk (iUP-201A model only).
COPY (buffer to URAM)	Loads the buffer into the iUP URAM (iUP-201A model only).
COPY (URAM to buffer)	Loads iUP URAM data into the buffer (iUP-201A model only).
SECURITY GROUP	LOCKS SELECTED DEVICES TO PREVENT UNAUTHORIZED ACCESS.
KEYLOCK	Locks the EPROM from unauthorized access.

firmware for reading and programming a family of Intel devices. Each personality module is a single molded unit inserted into the front panel of the universal programmer. A wide variety of personality modules and adaptors are available for Intel programmable devices. New modules and adaptors allow you to keep abreast of the newest Intel devices, programming algorithms, and device packages while protecting your equipment investment. Refer to the data sheet on "PROM Programming Personality Modules" for a complete list of available support.

Each personality module connects to the universal programmer through a 41-pin connector. Module firmware is uploaded into the iUP RAM and executed by the internal processor. The personality module firmware contains routines necessary to read and program a family of EPROMs. In addition, the personality module sends specific information about the selected EPROM to the universal programmer to help perform EPROM device integrity checks.

Table 2. Off-Line Command Keys Summary

Key	Function
ON LINE	Selects either on-line or off-line operation. When on-line, all other function keys are disabled.
DEVICE SELECT	Selects the EPROM type when using a personality module able to program multiple EPROM devices.
VER	Verifies the contents of the installed EPROM device with the contents of the iUP RAM. The universal programmer display indicates the address and the XOR of any mismatches.
PROG	Performs a device blank check and then programs the target EPROM with data from the iUP RAM. If the blank check fails, pressing PROG again peforms an overlay check to verify that non-blank EPROMs can be programmed.
ROM TO RAM	Loads the iUP RAM with the data from the EPROM device installed in the personality module.
CLEAR	Terminates the current off-line function, clears a user entry, or restores the display after an error.
ENTER	Transfers information from the universal programmer display (addresses, data, or baud rate) into the iUP RAM.
SHIFT ADDR 0	Selects an address field for display.
SHIFT DATA 1	Selects a data field for keypad editing and entry.
SHIFT FILL 2	Loads a contiguous section of iUP RAM locations with a constant.
SHIFT LOAD 3	Downloads Intel hexadecimal data from any development system which has an RS232C port.
SHIFT LOCK 4	Locks a EPROM from unauthorized access.

210319-8

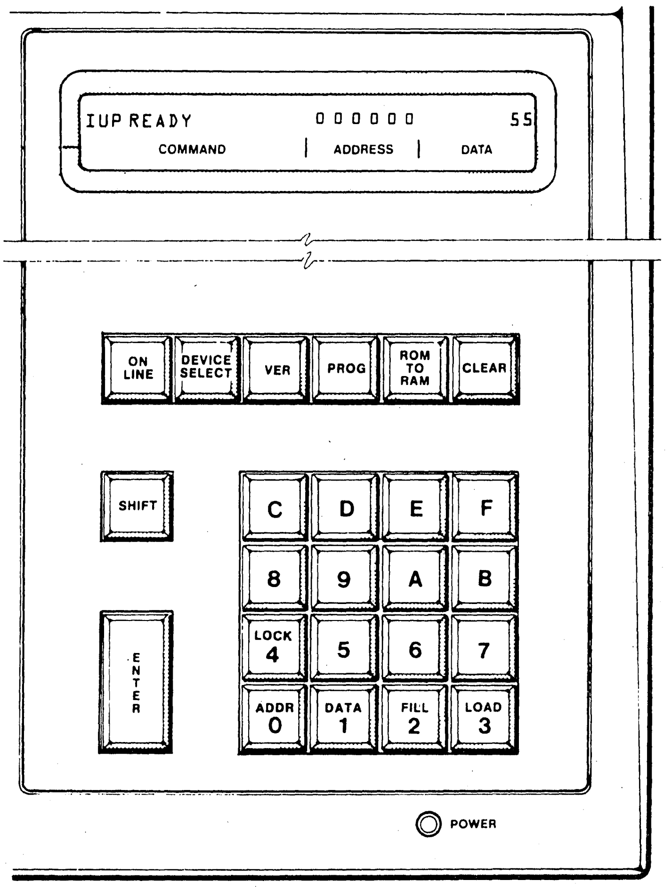


Figure 3. iUP-201A Keyboard and Display

LEDs on each personality module indicate operational status. On some personality modules a column of LEDs indicate which EPROM device type the user has selected. On some personality modules an LED below the socket indicates which socket is to be used. A red indicator light tells the user when power is being supplied to the selected device. Figure 5 shows a selection fo some of the personality modules supported on the universal programmer.

In addition to the testing done by the iUP system self-tests, each personality module contains diagnostic firmware that performs selected EPROM tests and indicates status. These tests are performed in both on-line and off-line modes. The EPROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed. The EPROM blank check determines whether a device is blank. The universal programmer automatically determines whether the blank state is all zeros or all ones. The overlay check (performed when a EPROM is not blank) determines which bits are programmed, compares those bits against the program to be loaded, and allows programming to continue if they match. As with the system self-tests, straight-forward messages are provided. The user can invoke all of the EPROM device integrity checks except the installation test (which occurs automatically any time an operation is selected).

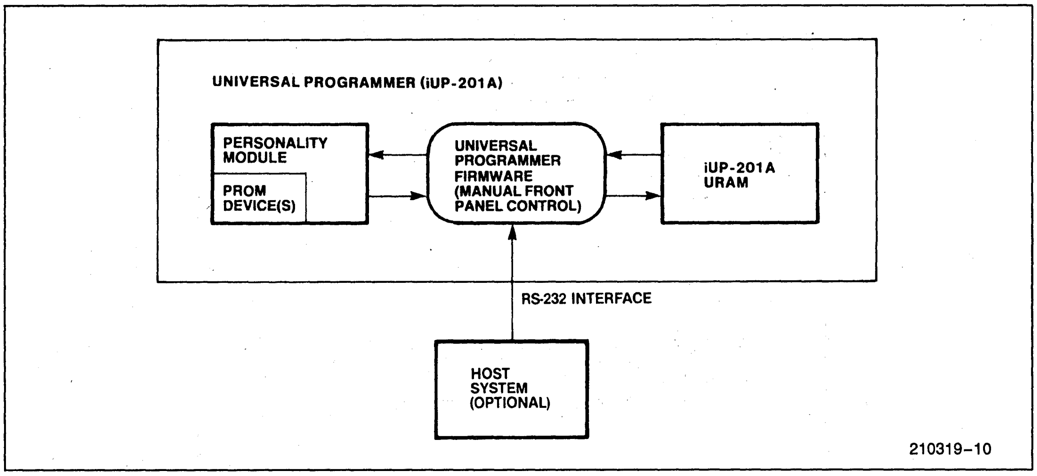


Figure 4. Off-Line System Data Flow

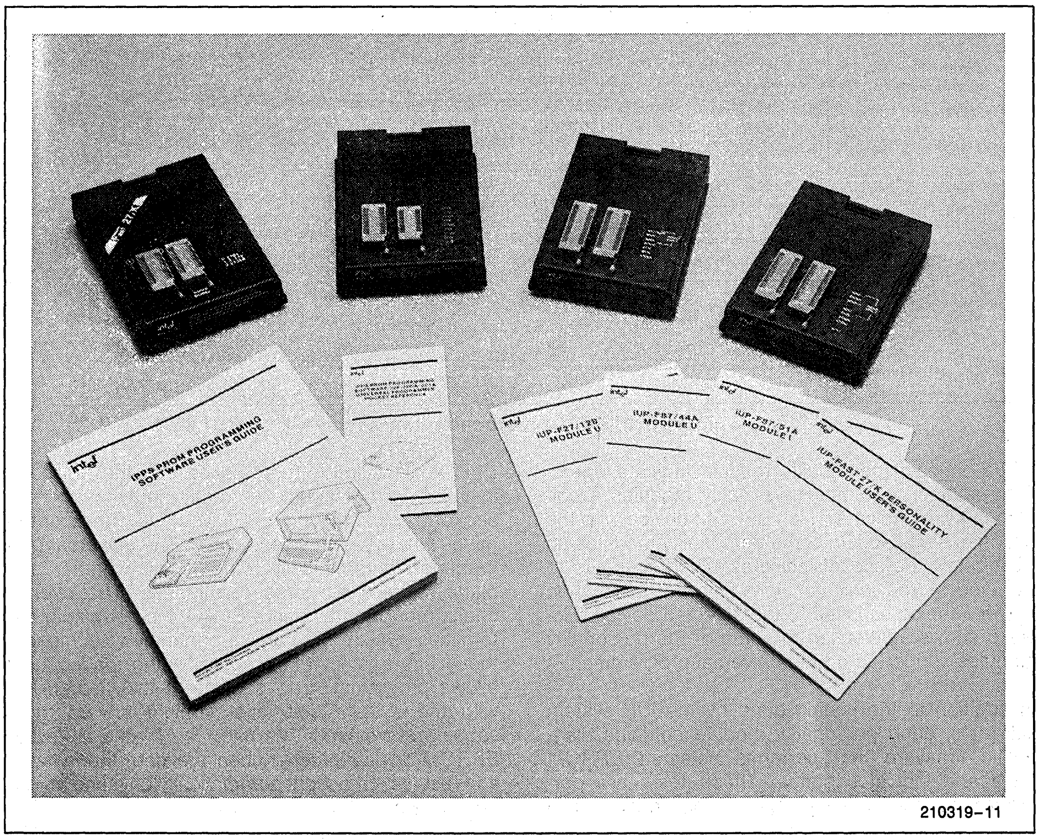

iPPS PROM PROGRAMMING SOFTWARE USER'S GUIDE
iUP-FAST 27/K PERSONALITY MODULE USER'S GUIDE
210319-11

Figure 5. Personality Modules

iUP-200A/iUP201A SPECIFICATIONS

Control Processor

Intel 8085A microprocessor
6.144 MHz clock rate

Memory

RAM—4.3 bytes static
ROM—12K bytes EPROM

Interfaces

Keyboard: 16-character hexadecimal and 12-function keypad (iUP-201A model only)

Display: 24-character alphanumeric (iUP-201A model only)

Software

Monitor— system controller in pre-programmed EPROM

iPPS — Intel PROM programming software on supplied diskette

Physical Characteristics

Depth: 15 inches (38.1 cm)
Width: 15 inches (38.1 cm)
Height: 6 inches (15.2 cm)
Weight: 15 pounds (6.9 kg)

Electrical Characteristics

Selectable 100, 120, 200, or 240 Vac ± 10%; 50-60 Hz

Maximum power consumption—80 watts

Environmental Characteristics

Reading Temperature:	10°C to 40°C
Programming Temperature:	25°C ±5°
Operating Humidity:	10% to 85% relative humidity

Reference Material

166041-001— *iUP-200A/201A Universal Programmer User's Guide.*

166042-001— *Getting Started with the iUP-200A/201A (For ISIS/iNDX Users).*

166043-001— *Getting Started with the iUP-200A/201A (For DOS Users).*

164853 — *iUP-200A/201A Universal Programmer Pocket Reference.*

ORDERING INFORMATION

Product Order Code	Description
iUP-200A 211A	On-Line PROM programmer with iPPS rel 1.4 on Single density ISIS II floppy
iUP-200A 212B	On-Line PROM programmer with iPPS rel 1.4 on Double density ISIS II floppy
iUP-200A 213C	On-Line PROM programmer with iPPS rel 2.0 for Series IV, on mini-floppy
iUP-200A 216D	On-Line PROM programmer with iPPS rel 2.0 for PC/DOS, and cable for PC or XT
iUP-200A 217D	On-Line PROM programmer with iPPS rel 2.0 for PC/DOS, and cable for AT
iUP-201A 211A	Off-Line and on-line PROM programmer with iPPS rel 1.4 on Single density ISIS II floppy
iUP-201A 212B	Off-Line and on-line PROM programmer with iPPS rel 1.4 on Double density ISIS II floppy
iUP-201A 213C	Off-Line and on-line PROM programmer with iPPS rel 2.0 for Series IV on mini-floppy
iUP-201A 216D	Off-Line and on-line PROM programmer with iPPS rel 2.0 for PC/DOS, and cable for PC or XT
iUP-201A 217D	Off-Line and on-line PROM programmer with iPPS rel 2.0 for PC/DOS, and cable for AT
iUP-200/201 U1* Upgrade Kit	Upgrades an iUP-200/201 universal programmer to an iUP-200A/201A universal programmer
iUP-PAK-A Upgrade Kit	Upgrades an iUP-200/A universal programmer to an iUP-201A universal programmer

*Most personality modules can be used only with an iUP-200A/201A universal programmer or an iUP-200/iUP201 universal programmer upgraded to an A with the iUP-200/iUP-201 U1 upgrade kit. If used in an iPDS, most personality modules require version 1.4 of the iPPS software.

Product Order Code	Description
iUP-PAK-A Upgrade Kit	Upgrades an iUP-200A universal programmer to an iUP-201A universal programmer

Software Sold Separately

Product Order Code	Description
211A	PROM programming software rel 1.4 on Single density ISIS II floppy
212B	PROM programming software rel 1.4 on Double density ISIS II floppy
213C	PROM programming software rel 2.0 for Series IV on mini-floppy
216D	PROM programming software rel 2.0 for PC/DOS with cable for PC or PC XT
217D	PROM programming software rel 2.0 for PC/DOS with cable for PC AT
219F	PROM programming software for iPDS on mini-floppy

iUP/iPDS™ PROGRAMMING MODULES

MAJOR PERSONALITY MODULE FEATURES:

- **Fast Support for All Intel EPROM and EPLD Device Types**
- **Adapts an iUP-200A/iUP-201A Universal Programmer or Intel Personal Development System (iPDS™) to a Family of PROM Devices**
- **The Fast 27/K-CON Kit Adds the Quick-Pulse Programming™ Algorithm to the Fast 27/K—the Fastest in the Industry**
- **Includes the iUP-GUPI Module with New Low-Cost Plug-In Adaptors for Programming Intel's Newest Devices**
- **Program Intel or Intel-Compatible Devices, Including Microcontrollers, EPLDs, CMOS EPROMs, Latched EPROMs, and the New Page-Programmable 27011 One Meg EPROM**

Personality modules custom-fit the iUP-200A/iUP-201A Universal Programmer or the iPDS™ system to a family of PROM devices. Each personality module comes ready to use—just plug it into a Universal Programmer or an iPDS system and begin reading or programming parts. The personality modules can be used off-line or controlled from a host or iPDS system using Intel's powerful PROM programming software (iPPS). Selected personality modules support the latest PROM programming features such as the int$_e$ligent Programming™ algorithms (reduce programming time up to a factor of 10), the int$_e$ligent Identifier™ (automatically selects the correct int$_e$ligent Programming algorithm), and the security bit function (protects PROM memory from unauthorized access).

280003-1

*IBM Personal Computer is a registered trademark of International Business Machines Corporation.

October 1986
Order Number: 280003-003

PROGRAMMING MODULE DESCRIPTION

The personality module and GUPI module adapts the universal programmer or the iPDS system to a specific family of PROM devices; it contains all the hardware and firmware necessary to read and program a family of Intel PROMs. The module comes ready to use; the user merely inserts the module into the universal programmer front panel or the side door of the iPDS chassis (refer to Figures 1 and 2).

Each module connects to the universal programmer/iPDS system through a 41-pin connector. LEDs on the module indicate its operational status. A column of LEDs or a hexadecimal display indictes which PROM device type the user has selected. On some modules, an LED below the socket indicates which socket is to be used. A red indicator light tells the user when power is applied to the selected device.

After specifying the PROM device type, the user inserts the PROM to be programmed or read in the socket on the module. The module checks for correct PROM installation. In addition, each module contains diagnostic firmware that performs the following selected PROM tests and indicates status.

- The PROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed.
- The PROM blank check determines whether a device is blank. The universal programmer/iPDS system automatically determines whether the blank state is all zeros or all ones.
- The overlay check (performed when a PROM is not blank) determines which bits are programmed, compares those bits with the program to be loaded, and allows programming to continue if they match.

The user can invoke all the PROM device integrity checks except the installation test (which occurs automatically any time an operation is selected).

iUP-GUPI* MODULE DESCRIPTION

The iUP-GUPI is a generic module that enables the iUP-200A/201A Universal Programmer and the iPDS system to accept low-cost plug-in adaptors. These adaptors configure the system to support a wide variety of programmable devices (EPROMs, microcontrollers, and EPLDs) and device package types (refer to Table 1).

The iUP-GUPI module plugs into any compatible Intel PROM programmer (the iUP-200A/201A Universal Programmer or the iPDS system). An opening in the top of the iUP-GUPI is provided for easy plug-in installation of the GUPI adaptors (refer to Figure 3).

***NOTE:**
Generic Universal Programmer Interface.

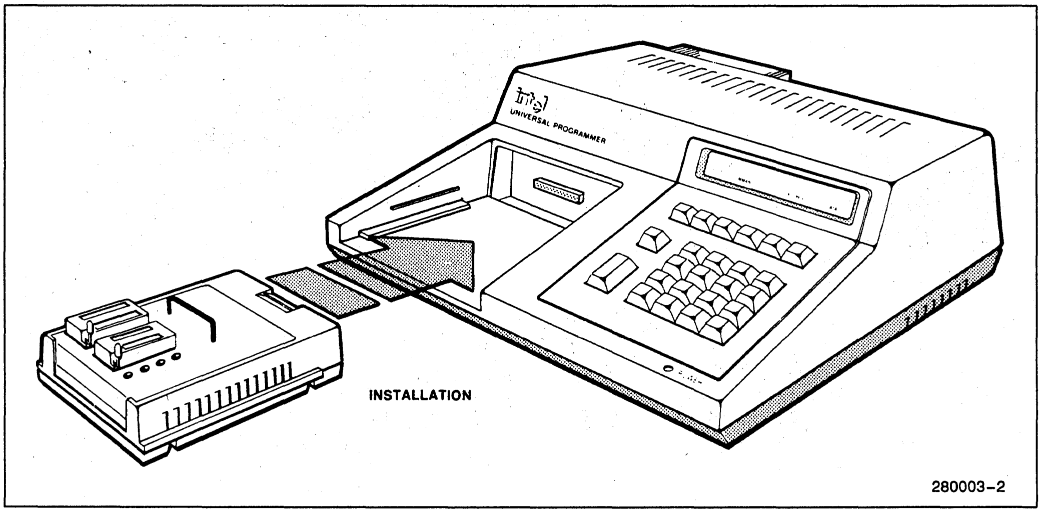


Figure 1. iUP-201A Universal Programmer

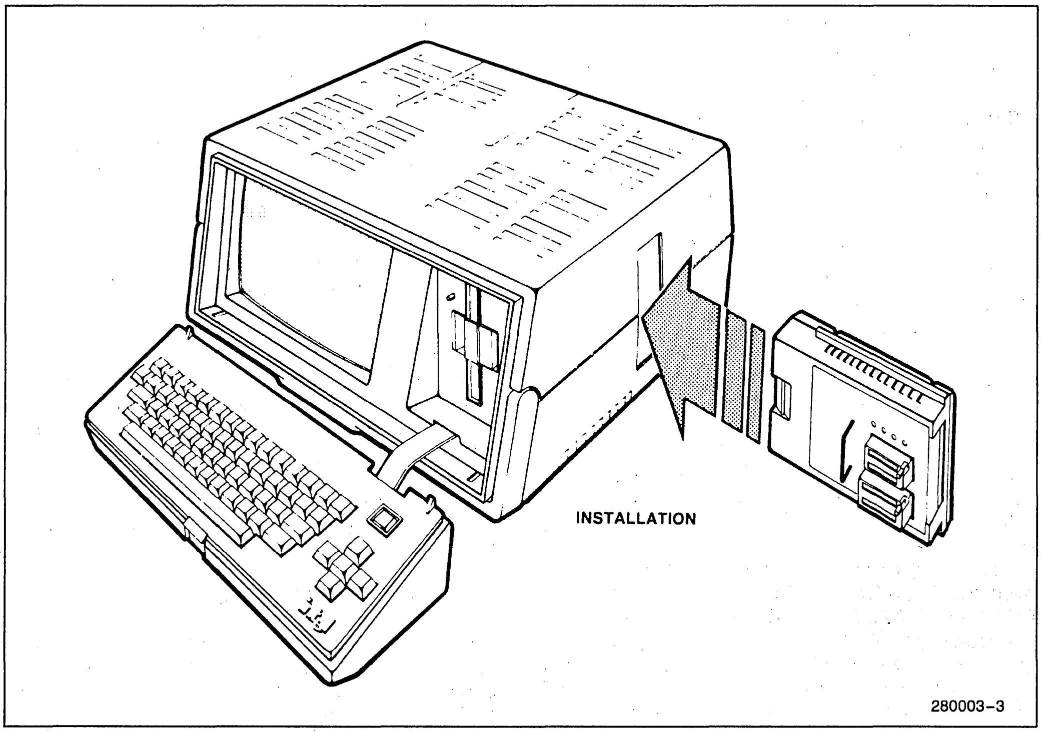


Figure 2. iPDS™ System

Table 1. GUPI Module Adaptors

Device Type	GUPI Logic-09	GUPI Logic-12	GUPI Logic-18	GUPI 27010	GUPI 27011	GUPI 27210	GUPI 8742	GUPI* 8796
EPLD	5C060 5C090	5C031 5C121	5C180					
EPROM				27010	27011	27210		
Microcontroller							8741AH 8742AH 8041AH 8042AH	8794 8795 8796 8797

*For Pin Grid Array (PGA) and CERDIP packages.

The iUP-GUPI offers all of the same programming performance as earlier personality modules, with the addition of employing Intel's latest, fastest programming algorithms and providing support for several different device types. For example, the iUP-GUPI uses the new Quick-Pulse Programming™ algorithm to program the 1 Meg EPROM in seconds. The initial set of GUPI adaptors and devices supported are listed in Table 1. More adaptors will be announced in the future supporting additional devices and package types.

iUP-GUPI and GUPI LOGIC Adaptors

The iUP-GUPI and assorted GUPI LOGIC adaptors provide an alternative programming solution for Intel's H-series and Altera EPLD devices, when purchased with the iPLS, Intel's Programmable Logic Software. This complete set of software is available separately (i.e., without the iLP programmer pod and IBM interface card).

By selecting a system consisting of the iPLS software, iUP-201A (with iPPS software for the IBM PC, PC XT, or PC AT), and iUP-GUPI, no expansion slots are used in your PC (since the iUP communicates via the PC's RS232 serial port), and a more versatile programming solution is obtained. Some of the added programming advantages are stand-alone operation when several duplicate EPLDs are needed, increased device testing with checksum, verification, and optional programming of EPROMs and microcontrollers with low cost adaptors.

PROM PROGRAMMERS

The modules are used with either the universal programmer or the iPDS system as illustrated in Figure 4. Both the iUP-200A and iUP-201A models of the universal programmer program PROM devices in on-line mode. The iPPS software which controls on-line programming, runs on a variety of host systems. The iUP-201A universal programmer adds an additional feature: off-line programming directly from the universal programmer's keyboard. Figure 1 shows an iUP-201A universal programmer with a module inserted.

The iPDS system features stand-alone on-line programming controlled by the iPDS-iPPS software which runs on the iPDS system. The iPDS system operates in on-line mode only. Figure 2 shows an iPDS system with a module inserted.

Table 2 compares the features of the universal programmer with the features on the iPDS system.

THE iPPS SOFTWARE

The iPPS software, included with both the iUP-200A and iUP-201A models of the universal programmer and with the iPDS system, brings increased flexibility to PROM programming. The iPPS software provides user control through an easy-to-use interactive interface and performs the following functions to make PROM programming quick and easy:

- Reads PROMs and ROMs.
- Programs PROMs directly or from a file.
- Verifies PROM data with buffer data.

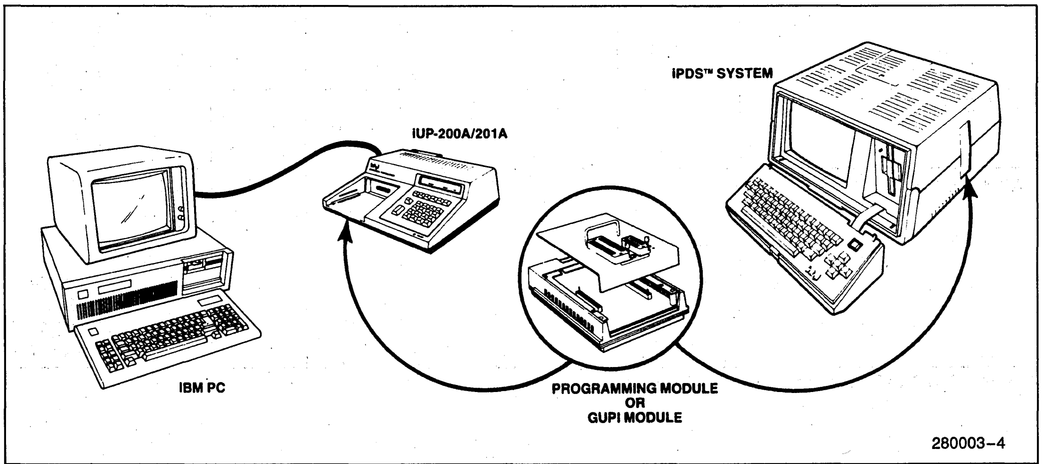


Figure 3. System Configuration

Table 2. PROM Programmers

Features	iUP-200A Universal Programmer	iUP-201A Universal Programmer	IPDS™ System
Function	PROM Programmer	PROM Programmer	Development System and PROM Programmer
Operating Mode	On-Line Mode	On-Line Mode and Off-Line Mode	On-Line Mode
Configuration	Requires Host System Running iPPS Software	Requires Host System in On-Line Mode; Stand-Alone in Off-Line Mode	Stand-Alone Plugged Into iPDS System
Data Display	On CRT of Host System Terminal	On Built-In Single-Line Display in Stand-Alone Mode	On iPDS System CRT
Input Keyboard	From Host System Terminal	Built-In Keyboard	From iPDS System Keyboard

- Locks EPROM memory from unauthorized access (on devices which support this feature).
- Prints PROM contents on the network printer (universal programmer only) or the development system printer.
- Performs interactive formatting operations such as interleaving, nibble swapping, bit reversal, and blocks moves.
- Programs multiple PROMs from the source file, prompting the user to insert new PROMs.
- Uses a buffer to change PROM contents.

With the iPPS software the user can load programs into a PROM from system memory or directly from a disk file. Access to the disk lets the user create and manipulate data in a virtual buffer. This block of data can be formatted into different PROM word sizes for program storage into several different PROM types. In addition, a program stored in the target PROM, the system memory, or a system disk file can be interleaved with a second program and entered into a specific target PROM or PROMs.

The iPPS software supports data manipulation in the following Intel formats: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, 80286 absolute object, and 80386 bootloadable object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. The user can easily change default data formats as well as number bases.

Versions of the iPPS software are available to run on a variety of host microcomputers and operating systems, including the Intel ISIS/iNDX and IBM PC-DOS operating systems. Contact your local field sales office for a complete list of compatible hosts.

PERSONALITY MODULE FEATURES

The personality modules described in the following sections enable a universal programmer/iPDS system to program a wide range of PROM devices, each with its unique needs and requirements: PROMs, EPROMs, E^2PROMs, microcontrollers, and microprocessor peripherals. Refer to Table 3 for a comparison of the devices supported by each module.

Note that the user needs one of the following configurations to use the Fast 27/K personality module or to use the security bit function on the iUP-F87/51A and iUP-F87/44A personality modules:

- iPDS System
  - — Intel PROM programming software (iPPS-iPDS), version 1.4 or later
  - — iPDS-140 EMV/PROM adapter option
- universal programmer
  - — on-line
    Intel PROM programming
    software (IPPS),
    version 1.4 or later
    model 200A or 201A
    model 201A
  - — off-line

The user can easily update an earlier iUP-200/201 universal programmer to an iUP-200A/201A universal programmer with the iUP-200/201 U1 upgrade kit.

Table 3. iUP/PDS™ Programming Modules

PROM Type	Fast 27/K Module	Fast 27/K U2 Kit	Fast 27/K-CON* Kit	F27/128 Module	F87/44A Module	F87/51A Module
EPLD				2716		
				2732		
				2732A		
	2764	2764	2764	2764		
	2764A	2764A	2764A			
		27C64	27C64			
		87C64	87C64			
	27128	27128	27128	27128		
		27128A	27128A			
	27256	27256	27256			
		27C256	27C256			
		27512	27512			
		27513	27513			
KEPROM		27916				
E2PROM				2815		
				2816		
		2817A	2817A			
Microcontroller					8041A	8748
					8042	8748H
					8044AH	
					8741A	8749H
					8742	8751
					8744H	8751H
						8048
						8048H
						8049
						8049H
						8050H
						8051
					8755A	

*NOTE:
Quick-Pulse Programming algorithm.

The iUP-Fast 27/K Personality Module

With the iUP-Fast 27/K personality module the user can program, read, and verify the contents of Intel's newest EPROMs, from the page-programmable (512K) 27513, to the keyed-access 27916, to the CMOS 27C64, 27C256, and 87C64 EPROMs. This personality module supports the int$_e$ligent Programming algorithms and the int$_e$ligent Identifier. The int$_e$ligent Identifier lets the personality module interrogate the PROM device in the program/master socket. It determines whether the type selected matches the type of PROM device installed and then selects the proper int$_e$ligent Programming algorithm. The int$_e$ligent Programming algorithms reduce programming time up to a factor of 10.

Low cost, plug-in upgrade kits allow addition of support for Intel's latest EPROMs. The first upgrade kit added support for the 27512 and innovative page-programmable 27513 plus the 27128A and 2817A. It has now been replaced by a second upgrade kit, iUP-Fast 27/K-U2 adding support for Intel's new CMOS EPROMs and keyed-access KEPROM (refer to Table 3).

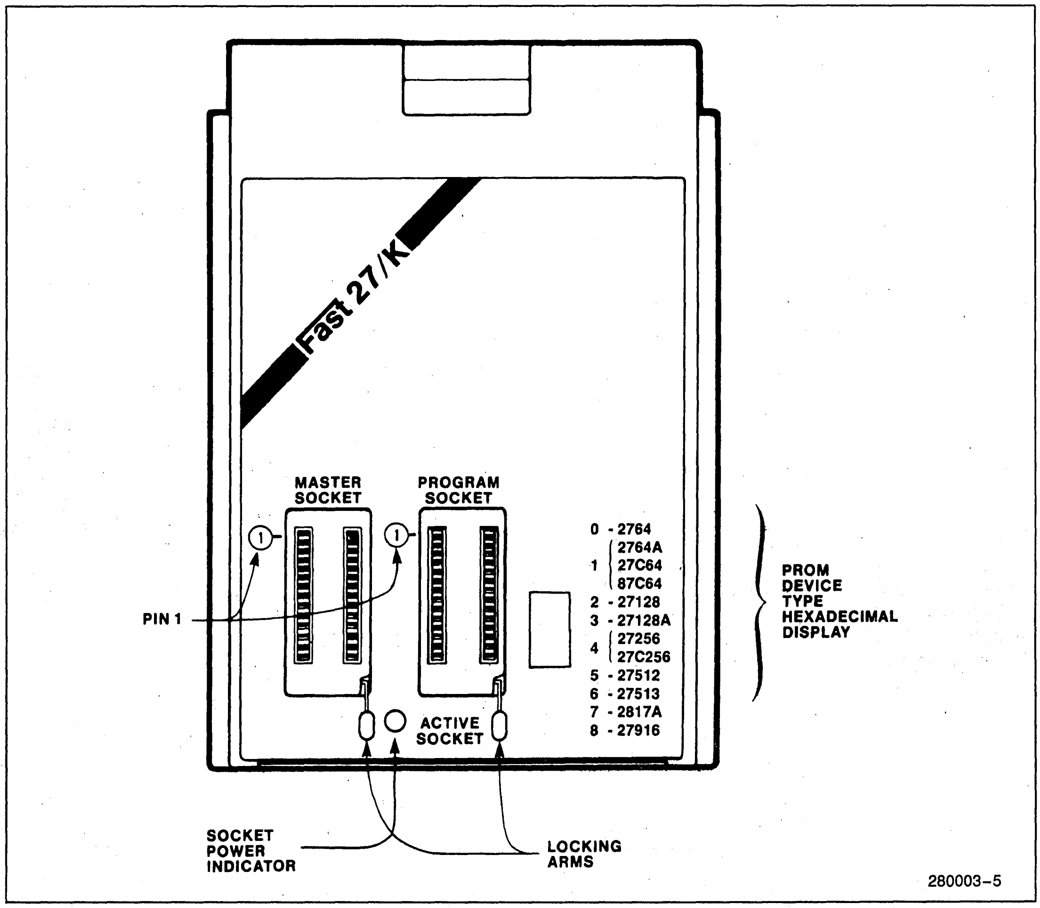


Figure 4. iUP-Fast 27/K Personality Module with U2 Upgrade

As shown in Figure 4, the iUP-Fast 27/K personality module contains two 28-pin sockets, a hexadecimal display (0 through F), and a red LED that indicates when power is being applied to a socket. The program socket holds the device being programmed. The master socket will be used in future upgrades. The hexadecimal display shows the PROM device type selected.

The iUP-F27/128 Personality Module

The iUP-F27/128 personality module lets the user program, read, and verify the contents of a wide variety of PROM devices, including some of Intel's most popular PROM devices (refer to Table 3).

The iUP-F27/128 personality module contains two sockets: one for 24-pin PROM devices and the other for 28-pin PROM devices. The user can use only one socket at a time. An LED below the socket indicates the correct socket to use based on the PROM device type selected, and a row of green LEDs on the right side of the personality module indicate which PROM type is selected. The ACTIVE SOCKET LED indicates when power is being applied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

The iUP-F87/51A Personality Module

The iUP-F87/51A personality module lets the user program EPROM microcontrollers and read the memory contents of ROM microcontrollers. This personality module supports the security bit function on the 8751H microcontroller. The KEYLOCK command locks the 8751H EPROM memory from unauthorized access by setting the security bit (which cannot be unlocked without erasing the device). As a safety precaution, the KEYLOCK command requires user verification before locking the security bit (refer to Table 3).

The iUP-F87/51A personality module has two sockets for inserting applicable PROM devices: one for the MCS®-48 family of devices and the other for the MCS-51 family of PROM devices. An LED below the socket indicates the correct socket to use based on the PROM device type selected. One of the green LEDs on the right side of the personality module lights to indicate the PROM type selected. The ACTIVE SOCKET LED lights when power is applied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

The iUP-F87/44A Personality Module

The iUP-F87/44A personality module lets the user program EPROM versions of the 8044 family of microcontroller/serial interface units and read the memory contents of ROM versions (refer to Table 3). This personality module supports the security bit function on the 8744H microcontroller. The KEYLOCK command locks the 8744H EPROM memory from unauthorized access by setting the security bit (which cannot be cleared without erasing the device). As a safety precaution, the KEYLOCK command requires user verification before setting the security bit.

The iUP-F87/44A personality module has two sockets for inserting applicable PROM devices: one for the 8741A, 8742, and 8755A PROM devices and the other for the 8744H PROM device. An LED below each socket indicates the correct socket to use based on the PROM device type selected. One of the green LEDs on the right side of the personality module lights to indicate the PROM type selected. The ACTIVE SOCKET LED lights when power is ap plied to the PROM device and when the universal programmer/iPDS system is accessing the selected socket.

PROM PROGRAMMING EXAMPLE

The personality module is the interface that lets the user perform a wide variety of PROM programming, data display, and data editing operations. One of the most popular applications is copying data from a master PROM into a blank PROM. Table 4 outlines and compares the steps for both on-line and off-line copying. Notice the easy-to-use, English-language approach of the iPPS commands, which may be shortened to the first letter for faster entry.

The on-line example assumes that the universal programmer/iPDS system has been powered on and is under control of the ISIS software and that the iPPS software has been initialized. The off-line example assumes that the iUP-201A universal programmer has been powered on and initialized.

PERSONALITY MODULE SPECIFICATIONS

Physical Characteristics

Width: 5.5 inches (1.4 cm)
Height: 1.6 inches (4.1 cm)
Depth: 7.0 inches (17.8 cm)
Weight: 1 pound (0.45 kg)

Electrical Characteristics

Maximum power consumption (module): 7.5 watts
Maximum power consumption (device): 2.5 watts
Maximum power consumption (total from PROM programmer): 10 watts

Table 4. Typical PROM Programming Sequence

Action	On-Line Command	Off-Line Function Key
1. Select PROM type.	TYPE	DEVICE SELECT
2. Install the PROM to be copied (the master PROM) in the personality module.		
3. Copy the contents of the master PROM to the buffer.	COPY PROM TO BUFFER	ROM TO RAM
4. Verify that the copy was correct.	VERIFY	VER
5. Remove the master PROM; install a blank PROM.		
6. Copy the buffer to the blank PROM.	COPY BUFFER TO PROM	PROG

Environmental Characteristics

Reading temperature: 10°C to 40°C
Programming temperature: 25°C ±5°
Operating humidity: 10%–85% relative humidity

DOCUMENTATION

Appropriate personality module user's guide:

164376— *iUP-FAST 27/K Personality Module User's Guide*

165833— *iUP-FAST 27/K-U2 Upgrade Kit Installation Manual*

162848— *iUP-F27/128 Personality Module User's Guide*

164855— *iUP-F87/51A Personality Module User's Guide*

164854— *iUP-F87/44A Personality Module User's Guide*

166428— *iUP-GUPI Module User's Guide*

ORDERING INFORMATION

Part Number	Description
iUP-FAST 27K*	EPROM personality module
iUP-FAST 27/K-U2	Upgrade Kit
iUP-F27/128	EPROM and E^2PROM personality module
iUP-F87/51A*	Microcontroller personality module
iUP-F87/44A*	Peripheral personality module
iUP-FAST 27/K-CON	Upgrade Kit
iUP-GUPI	Generic module interface
GUPI LOGIC-09	Adaptor
GUPI LOGIC-18	Adaptor
GUPI-27010	Adaptor
GUPI-27011	Adaptor
GUPI-27210	Adaptor
GUPI-8742	Adaptor
GUPI-8796	Adaptor

***NOTE:**
The iUP-FAST 27/K personality module and the security bit function on the iUP-F87/51A and iUP-F87/44A personality modules can be used with an iUP-200A/201A universal programmer; or an iUP-200/iUP-201 universal programmer upgraded to an A with the iUP-200/201 U1 upgrade kit; or an iPDS system, using version 1.4 or later of the iPPS-iPDS software (iPDS-140 units shipped after June 1984 contain this software).

intel®

APPLICATION NOTE

AP-179

October 1986

PROM Programming with the Intel Personal Development System (iPDS™)

FRED MOSEDALE
DSHO TECHNICAL PUBLICATIONS

Order Number: 280015-001

- **Need for simple operation**—You want a programming device that satisfies all of the following needs and is simple to operate. You do not want to have to refer to a manual every time you wish to program a PROM.
- **Need to program a wide variety of PROMs**—For greatest flexibility, you want a programming device that can program the various kinds of PROMs that are available. For example, you will want to be able to program microcontrollers with EPROMs, and you will want to be able to program from the small inexpensive 16K and 32K PROMS to the latest 256K PROMS with int$_e$ligent Programming™ algorithms to speed programming. You will also want to be able to program those PROMs that use the new lower programming voltage (12.5V).
- **Need to be upgradeable**—A PROM programming device should be designed so that it can be upgraded to program PROMs that will be available in the future. Without upgradeability, the device will soon be out-of-date.
- **Need to check PROM contents before programming**—If the PROM you will be programming is blank, it can of course be programmed. Even if it has some bits set at the time of programming, if the same bits must also be set for the program, the PROM can be used. Thus, ideally, a PROM programming device will determine whether the PROM is blank, and if not, determine whether the bits already set are compatible with the program to be loaded into the PROM.
- **Need to recognize file formats**—When a PROM is programmed using an object file generated by a compiler or assembler, the PROM programming device must be able to extract the data that is to be loaded into the PROM from the larger file structure. For greatest flexibility, you will need a PROM programming device that recognizes the file structures generated by compilers and assemblers that you will use when developing program code for the PROMs.
- **Need to support a variety of source program options**—There are three sources you may wish to use for PROM data: an already-programmed PROM, a software development system, or a disk. For greatest programming flexibility, you will want PROM programming device that makes all three kinds of sources available.
- **Need to support data manipulation and modification**—You may wish to modify a source file for your PROM program. For example, you may discover an error in the source file, or you may realize that for your new processor system, the PROM data must first be 2's complemented. For a variety of reasons, your PROM programming will be much more flexible if the PROM programming device offers a buffer for temporary storage and PROM programming software that can manipulate the source program data in a variety of ways.
- **Need for variety in loading program code into PROMs**—If your product has a 16-bit microprocessor and you are using 8-bit PROMs to store the firmware, you will need to interleave the 16-bit code between two 8-bit PROMs. A PROM programming device with interleaving capability will speed such programming. You may want other kinds of flexibility when programming.
- **Need to transfer long programs that will not fit into one PROM**—Long programs may exceed the storage capacity of the PROMs chosen for your product. You need a programming device that can format the program so that it can be stored in successive PROMs.
- **Need to verify programming**—When programming is finished, you will want to check that the PROM is indeed correctly programmed. A defect in the PROM could corrupt the intended firmware. Checking would involve comparing the source with the programmed PROM.
- **Need to compare buffer with PROM**—If you are interrupted when programming PROMs or if you have not labeled PROMs that you did program, you may forget whether a particular PROM was programmed. In such cases, you will want to compare the particular PROM with the program stored in the buffer of the programming device. Comparison will prevent you from having to reprogram an already-programmed PROM.

INTRODUCTION

Programmable read-only memory (PROM) devices play a significant role in microprocessor-based products. How can PROM programming devices perform to best serve the needs of those who develop and service such products?

This application note first provides a general answer to this question; then, it proceeds to describe the features and use of PROM programming hardware and software available for the Intel Personal Development System (iPDS™). The description explains how the iPDS system provides the wide range of capabilities needed by those who use PROM programming devices. The description also highlights the iPDS system's ability to program some of Intel's newest EPROMs.

PROMS IN THE LIFE CYCLE OF A PRODUCT

Memory Options: A Review

Before PROM programming needs are discussed, it is important to briefly review memory options available to designers.

Microprocessor-based products need memory to store instructions and data used in controlling their operations. In order to maximize product operating speeds, designers must use memory that can be accessed quickly. Both random access memory (RAM) devices and read-only memory (ROM) devices offer designers quick access, but RAM devices are volatile—their contents are erased when system power is turned off. ROM devices are nonvolatile; thus designers use ROMs to store programs and data that will not change during the operation of the product.

After a product's program code data has been debugged, you can transfer the code to programmable ROMs (PROMs) or masked ROMs. (The most flexible PROMs are E^2PROMs and EPROMs; E^2PROMs are electrically erasable and EPROMs can be erased by ultraviolet light.) PROMs are programmed using a relatively simple procedure; by contrast, masked ROMs can only be programmed in a manufacturing environment. Thus, masked ROMs provide less flexibility but are used because they may be more cost-effective in large volumes. However, because the price of PROMs is falling and because inventories with erasable PROMs can be reprogrammed when product programs are changed, erasable PROMs are also attractive for large volumes.

Desirable PROM Programming Features

Once your product's software is debugged, you can load the software into the product's PROMs (so that it becomes the product's firmware). However, usually in the development of a product, the initial programming of PROM devices is not the last operation involving the PROMs. Even if the software is debugged, once it is loaded into the PROMs, you may discover new bugs in the program that you failed to detect before the program was committed to PROMs. So, there may soon be a need to erase the PROMs (if they are EPROMs or E^2PROMs) and reprogram them.

During product development and servicing, you will also sometimes need to accomplish the following tasks:

- Check the contents of a PROM.
- Use one PROM to program other PROMs that will be used in other prototype systems.
- Update earlier firmware versions with later versions.

Consider in more detail the (P)ROM-related needs that can arise for you during the product's life cycle, that is, between the time when the product's software has been loaded into PROMs and the time when the product is phased out. There are two basic sets of needs, those having to do with displaying (P)ROM contents and those having to do with programming PROMS.

DISPLAYING AND PRINTING NEEDS

For a variety of reasons, you may need to examine what is stored in a (P)ROM. For example, you may suspect that a (P)ROM's program is in error; or you may have incomplete documentation on what was programmed into a (P)ROM. Thus, you will want to be able to display the contents on a video display terminal and to have a printer print out the contents. You will want to be able to choose the display base (binary, octal, decimal, or hexadecimal) and whether to display the contents as ASCII characters.

PROGRAMMING NEEDS

When you program PROMs, you need a programming device that can program a vareity of PROMS and one that offers flexibility and ease of programming. The following list describes PROM programming needs.

- **Need to lock microcontrollers from unauthorized access**—Some advanced microcontrollers can be locked to prevent unauthorized access. To take advantage of this security features, you need to be able to control the locking of the microcontrollers.
- **Need to automate routine PROM programming tasks**—To speed programming, you will want a programming device that can automate routine programming functions. Automation will not only speed programming, it will also release personnel for other work.

DESIRABLE PROM PROGRAMMING FEATURES: A SUMMARY

In summary, if PROMs (and ROMs) are incorporated in your product, you will have the greatest flexibility if your PROM programming device has the following features. (Of course, for ROMs only reading tasks are needed.)

- Is easy-to-use.
- Can program a wide variety of PROMs.
- Is upgradeable.
- Can display (P)ROM contents in ASCII characters and a variety of bases.
- Can enable a printer to print out (P)ROM contents in ASCII characters and in a variety of bases.
- Can check for blank PROMs.
- If PROM is not blank, can check PROM contents for compatibility with program.
- Can recognize file formats of your development system object files.
- Supports transfers of program code from development systems, disks, and other PROMs to the new PROM.
- Provides temporary storage and software for manipulating Programming data before loading it into the PROM.
- Supports variety in how data is loaded into PROMs, e.g.:
  - — Interleaving 16-bit data into 8-bit PROMs.
  - — Segmenting long programs so that resulting program segments fit into successive PROMs.
- Can verify the accuracy of copying.
- Can compare programming buffer with PROM contents.
- Can lock microcontrollers from unauthorized access.
- Can automate routine PROM programming tasks.

The Intel Personal Development system (iPDS) with the PROM programming option meets these needs. The following sections describe the iPDS PROM programming hardware and software and show how this system can perform all of these tasks for a variety of PROMs.

THE iPDS™ PROM PROGRAMMING SYSTEM

The iPDS system supports integrated hardware and software development; it provides a complete set of software development tools and in-circuit emulators for hardware debugging and hardware-software integration. With its optional PROM programming hardware and software, the iPDS system also supports PROM programming.

Three components comprise the iPDS PROM programming system: the iPDS system (with the ISIS-PDS operating system and the plug-in module adapter board), the PROM programming modules, and the prom programming software. Each of these components is described briefly in the following sections.

iPDS™ System

To perform PROM programming tasks, the iPDS system must use its ISIS-PDS operating system and the plug-in module adapter board. The PROM programming software (iPPS-PDS) runs under the ISIS-PDS operating system.

The adapter board allows you to use both the PROM programming personality modules and emulation modules. It provides the interface between the modules and the iPDS system.

Figure 1 shows the IPDS system with a PROM programming personality module plugged into its side.

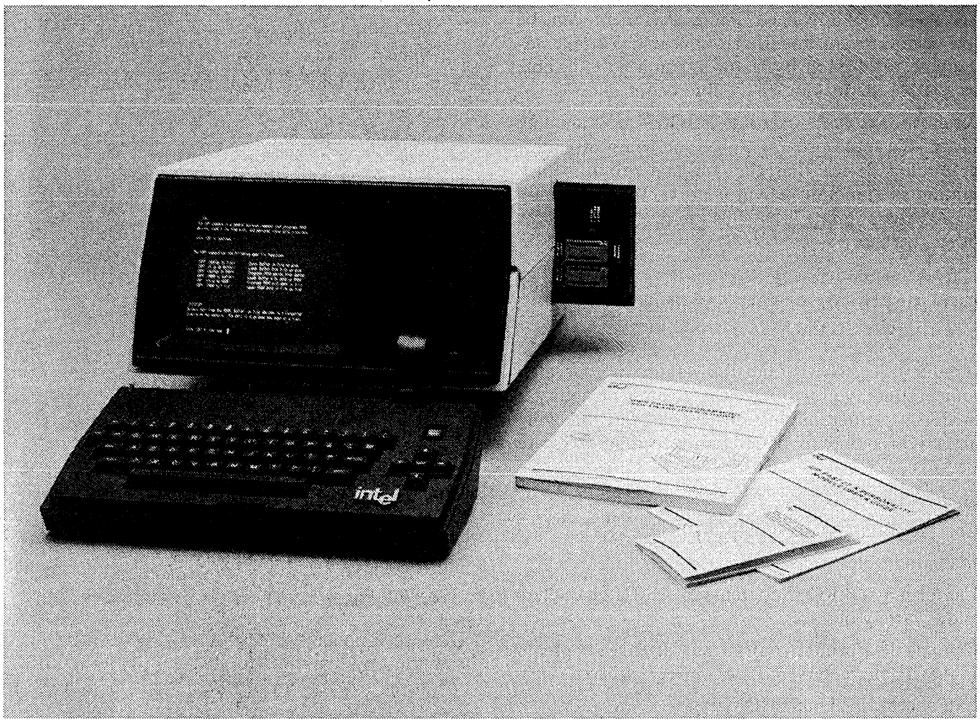


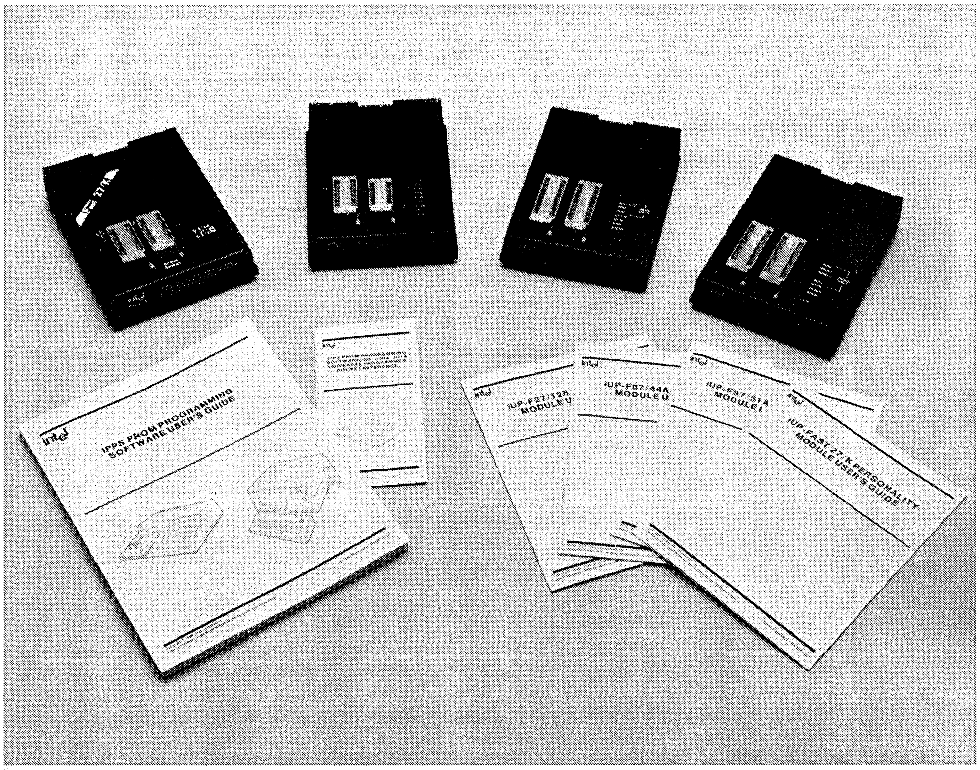
Figure 1. iPDS™ System with PROM Programming Personality Module

PROM Programming Personality Modules

A personality module is the interface between the iPDS system and a selected PROM. Personality modules contain all the hardware and firmware for reading and programming a family of Intel devices. Each personality module is a single molded unit inserted into the side panel of the iPDS unit. No additional adapters or sockets are needed. Table 1 lists the available personality modules, and Figure 2 shows the four modules.

Table 1. PROM Programming Personality Modules

Personality Module	PROM Type Programmed	PROMs and ROMs Supported
iUP-Fast 27/K	EPROM	2764, 2764A, 27128, 27256, and provisions for future PROMs.
iUP-F27/128	E^2/EPROM	2716, 2732, 2832A, 2764, 27128, 2815, and 2816
iUP-F87/51A	Microcontroller	8748, 8748H, 8048, 8749H, 8048H, 8049, 8049H, 8050H, 8751, 8751H, 8051
iUP-F87/44A	Peripheral	8741A, 8041A, 8742, 8042, 8744H, 8044AH, 8755A


iPPS PROM PROGRAMMING SOFTWARE USER'S GUIDE
iUP-F27/128 MODULE U
iUP-F87/44A MODULE U
iUP-F87/51A MODULE U
iUP-FAST 27/K PERSONALITY MODULE USER'S GUIDE

280015-2

Figure 2. PROM Programming Personality Modules

Each personality module connects to the iPDS system through a 41-pin connector. Module firmware is uploaded into the iPDS system and executed by the IPDS system. The personality module firmware contains routines needed to read and program a family of PROMs. In addition, the personality module sends specific information about the selected PROM to the iPDS system, such as information about the PROM size and its blank state.

LEDs on each personality module indicate its operational status. On some personality modules a column of LEDs or a hexadecimal display indicates which PROM device type the user has selected. On some personality modules with more than one socket, an LED below each socket indicates the socket to be used. In addition, a red indicator light tells the user when power is being supplied to the selected device.

The personality module firmware performs selected PROM tests and indicates status:

- The PROM installation test verifies that the device is installed in the module correctly and that the ZIF socket is closed.
- The PROM blank check determines whether the device is blank. The iPDS system automatically determines whether the blank state for the particular device is defined as all zeros or all ones.
- The overlay check (performed when a PROM is not blank) determines which bits are programmed, compares those bits against the program to be loaded, and allows programming to continue if they match.

Easy-to-read status messages are also provided. The user can invoke all of the PROM device integrity checks except the installation test (which occurs automatically any time an operation is selected). The following sections describe specific features of the three personality modules that program the newer Intel PROMs.

IUP-F87/44A AND IUP-F87/51A PERSONALITY MODULES: SPECIAL FEATURE

Each of these personality modules supports the security bit function on one member of the microcontroller family it can program. The iUP-F87/44A module supports the function on the 8744H microcontroller, and the iUP-F87/51A supports the function on the 8751H microcontroller. The KEYLOCK command locks the 8744H (or the 8751H) EPROM memory from unauthorized access by setting the security bit; the microcontroller cannot be unlocked without erasing the EPROM. As a safety precaution, the KEYLOCK command requires user verification before it sets the security bit.

IUP-FAST 27/K PERSONALITY MODULE: SPECIAL FEATURES

The iUP-Fast 27/K personality module supports the int$_e$ligent Identifier™ and the int$_e$ligent Programming algorithms. The int$_e$ligent Identifier is used to check the PROM installed in the personality module socket to determine whether it matches the type selected; then the int$_e$ligent Identifier is used to select the proper int$_e$ligent Programming algorithms. The int$_e$ligent Programming Algorithms reduce PROM programming time by as much as a factor of ten. This module has provision for support of future EPROMs and E^{2}PROMs via simple plug-in updates.

The int$_e$ligent Programming™ Algorithm

Using the capabilities of the IPDS PROM programming equipment and employing a new kind of algorithm that recognizes differences among EPROM cells, you can dramatically reduce programming time for the newest high-density EPROMs. As a bonus, the technique helps ensure that EPROMs receive adequate programming—in terms of memory-cell charge—to maintain long-term reliability.

Reducing programming time and costs for EPROMs has become increasingly important because the chips have become a cost-effective, easy-to-use alternative to masked ROM in high-volume applications requiring code flexibility or simplified inventory—a major switch from EPROMs' original small-volume prototyping applications. And, volume usage makes EPROM programming a significant manufacturing consideration.

The conventional programming procedure for most EPROMs uses a nominal 30-msec pulse per EPROM byte, resulting in a total programming time of approximately 1.5 minutes for a 16K-bit chip. With the introduction of the 2764 (64K bits) and devices with even higher densities, however, programming times have increased. A 256K-bit EPROM, for example, requires 24 minutes for programming using the conventional programming method.

Most EPROM cells program in less than 45 msec, however. In fact, empirical data shows that very few cells require longer than 8 msec for programming. Therefore, a procedure that takes into account the characteristics of individual EPROM cells can significantly reduce a device's programming time.

Arbitrarily reducing programming time is risky, however, because a cell's ability to achieve and maintain its programmed state is a function of this time. What is needed, therefore, is a way to verify the level to which individual cells have been programmed. Such a way exists. By determining the charge stored in a cell compared to the minimum charge needed to program the cell to a detectable level, you can check for a program margin that ensures reliable EPROM operation.

Margin checking does not occur in conventional EPROM programming, however. Instead, each EPROM cell receives a 45- to 55-msec write pulse, and manufacturers attempts to ensure program margin by screening out EPROMs having bytes that do not program within 45 msec. This programming procedure is thus an open loop—no actual verification of margin occurs.

By contrast, the int$_e$ligent Progrmaming algorithm guarantees reliability through the closed-loop technique of margin checking. This algorithm uses two different pulse types: initial and over-program. The algorithm first applies a 1-msec initial pulse to an EPROM. After the pulse, it checks the EPROM's output for the desired programmed value. If the output is incorrect, the algorithm repeats the pulse-and-check operation. When the output is correct, the algorithm supplies an over-program pulse; the length of this pulse depends on how many initial pulses were used and varies with the EPROM being programmed. This longer pulse helps ensure that the EPROM cell has an adequate programming margin for reliable operation.

Prom Programming Software (iPPS—PDS)

The iPPS-PDS software provides easy-to-use commands that allow you to load programs into a target PROM from another PROM, from iPDS system memory, or directly from a disk file.

The iPPS-PDS software also supports data manipulation in the following Intel formats: 8080 hexadecimal ASCII, 8080 absolute object, 8086 hexadecimal ASCII, 8086 absolute object, and 286 absolute object. Addresses and data can be displayed in binary, octal, decimal, or hexadecimal. You can easily change default data format as well as number bases.

You invoke the iPPS-PDS software from the ISIS operating system. (The software can be run under control of ISIS submit files, thereby freeing you from repetitious command entry.)

An explanation of the iPPS-PDS software follows. It is divided into three main sections: the iPPS-PDS storage devices, iPPS-PDS commands, and invoking the iPPS-PDS. Also see the Appendix for iPDS PROM programming examples.

iPPS-PDS STORAGE DEVICES

The iPPS-PDS software transfers data between any two of the three storage devices: PROM, buffer, and file. These devices are defined in the following three sections.

PROM Device

The PROM device is plugged into a socket on the personality module installed in the iPDS system. The iPPS-PDS software does not recognize the PROM device until you enter the TYPE command. The TYPE command automatically sets the appropriate buffer size according to the size of the PROM device specified.

Buffer Device

The buffer device is a section of development system memory that the iPPS-PDS software allocates and uses as a working area for temporary storage and for rearranging data. Its boundaries can exist anywhere in a virtual address range from 0 to 16777215 (0 to $2^{24}-1$).

When the iPPS-PDS software is initialized, the buffer starting address is set to 0, and the buffer ending address is set to 8K − 1, providing an initial buffer size of 8K bytes (the default buffer size when no PROM type is specified). During subsequent iPPS-PDS operations, the size and boundaries can vary. Specific iPPS-PDS commands determine these variations. The most recent command that changed the lower boundary of the buffer determines the buffer starting address. The TYPE command affects both the size and location of the buffer. For example, the TYPE command always resets the buffer start address to 0. The most recent TYPE command controls the size of the buffer.

The iPDS system needs a virtual buffer when PROM size exceeds 8K. If the PROM size exceeds the 8K memory buffer space available on the development system, the iPPS-PDS software creates a virtual buffer area using temporary file space on disk.

Two temporary work files are used to create the virtual buffer. During subsequent virtual buffer operations, the iPPS-PDS software automatically swaps data in and out of development system memory from and to work files.

File Device

The file device is an ISIS file on a disk. It is specified within iPPS-PDS commands.

The data stored in the disk file is in one of the following Intel absolute formats: 8080 hexadecimal, 8080 object, 8086 hexadecimal, 8086 object, or 80286 object. The iPPS-PDS software can read any of these formats as input but writes data to a file in 8080 object, 8086 object, or 80286 object formats only. Basically, these files contain representations of blocks of memory data. Included with the data are addresses for the locations of the data. The data blocks are not necessarily in consecutive address order. The method used to create the file determines the order of the data.

The iPPS-PDS file device has address boundaries that exist in the virtual range from 0 to 16777215 (0 to $2^{24}-1$). These boundaries are determined as follows:

- The file's lowest address is the lowest address encountered while reading the file.
- The file's highest address is the highest address encountered while reading the file.

If the iPPS-PDS software creates the file (that is, if the file is a destination device in an iPPS-PDS command), the specific command issued determines these boundaries.

When you specify a particular address range to be read from a file, all sections in the address range that are not present in the file are written in a PROM destination device as the blank state of the currently selected PROM type. If the destination device is the buffer, the nonexistent sections in the file do not overwrite the corresponding sections in the buffer.

During the operation of commands that use the file device as a source, the iPPS-PDS software only reads the actual data from the file and ignores any other information in the file. For example, the file can contain special information used later for debugging. Since the iPPS-PDS software ignores this information, it will not appear in any new files generated. If the data is written back to the original file, the original file is deleted.

iPPS-PDS COMMANDS

Each iPPS-PDS command consists of a keyword that identifies the command, followed by other keywords and associated parameters that are the arguments of the command. You enter all iPPS-PDS commands, as well as program address and data information, through the development system ASCII keyboard; the commands are displayed on the system CRT. Table 2 summarizes the iPPS-PDS commands.

Table 2. iPPS-PDS Command Summary

Command	Description
Program Control Group	**Controls Execution of the iPPS-PDS Software.**
EXIT	Exits the iPPS software and returns control to the ISIS operating system.
<ESC>	Terminates the current command.
REPEAT	Repeats the previous command
ALTER	Edits and re-executes the previous command.
Utility Group	**Displays User Information and Status; Sets Default Values.**
DISPLAY	Displays PROM, buffer, or file data on the console.
PRINT	Prints PROM, buffer, or file data on the local printer.
HELP	Displays user assistance information.
MAP	Displays buffer structure and status.
BLANKCHECK	Checks for unprogrammed PROMs.
OVERLAY	Checks whether non-blank PROMs can be programmed.
TYPE	Selects the PROM type.
INITIALIZE	Initializes default number base and file type.
WORKFILES	Specifies the drive device for temporary work files.
Buffer Group	**Edits, Modifies, and Verifies Data in Buffer.**
SUBSTITUTE	Examines and modifies buffer data.
LOADDATA	Loads a section of buffer with a constant.
VERIFY	Verifies data in the PROM with buffer data.
Formatting Group	**Rearranges Data from PROM, Buffer, or File.**
Format	Formats and interleaves buffer, PROM, or file data.
Copy Group	**Copies Data from One Device to Another.**
COPY (file to PROM)	Programs PROM with data in a file on disk.
COPY (PROM to file)	Saves PROM data in a file on disk.
COPY (buffer to PROM)	Programs PROM with data from the buffer.
COPY (PROM to buffer)	Loads the buffer with data in the PROM.
COPY (buffer to file)	Saves the contents of buffer in a file on disk.
COPY (file to buffer)	Loads the buffer from a file on disk.
Security Group	**Locks Selected Devices; Prevents Unauthorized Access.**
KEYLOCK	Locks the PROM from unauthorized access.

Once entered, a command line is verified for correct syntax and executed. If a syntax error is detected, the following error message is dispalyed:

```
--SYNTAX ERROR--specific error.
```

If you omit a required keyword, the iPPS-PDS software prompts for the keyword and its associated parameters: If the keyword is entered but its parameters are omitted, either a default value is assumed or an error message is displayed if there is no default. In certain commands, default keywords are also assumed.

You can enter complete iPPS-PDS keywords or any unique abbreviation (only the first character is required). For example, command keywords of C, CO, COP, and COPY are all interpreted as the COPY command.

The iPPS-PDS software accepts numeric entries in any one of four number bases: binary (Y), octal (O or Q), decimal (T), or hexadecimal (H). Numbers can be entered in any of these bases by appending the appropriate letter identifier to specify the base (e.g., 11111111Y, 377Q, 255T, FFH). An explicit number base identifier overrides the default number base, which is initially hexadecimal.

INVOKING iPPS

There are two methods of invoking the iPPS-PDS software: command lines and submit files.

The command line for invoking the iPPS-PDS software (under V1.0 and later versions of the ISIS.PDS operating system) uses the following syntax:

```
[:Fn:]IPPS
```

The symbol ":F*n*:" Specifies the drive on which the iPPS-PDS files are located. When you enter the iPPS-PDS command, the ISIS operating system loads and executes the iPPS-PDS software.

The iPPS-PDS software can also run under the control of a submit file. SUBMIT is an ISIS command that allows you to use a disk text file as input for further ISIS commands or as command inputs to utilities running under the ISIS operating system. Thus, a submit file can contain the ISIS command line to invoke the iPPS-PDS software and then a sequence of commands for the iPPS-PDS software itself.

Summary: the iPDS™ System Meets PROM Programming Needs

Table 3 describes briefly how the iPDS system meets each of the needs identified earlier in this application note.

The iPDS system can be a complete intelligent PROM programmer—and, because the iPDS system is also a development system, it can provide an excellent means to off-load PROM programming from your current development system (just as the iPDS system allows you to off-load other 8-bit development tasks). In addition, with its state-of-the-art PROM programming capability, the iPDS system becomes an attractive solution to your complete development system needs.

Table 3. iPDS™ Features Meet PROM Programming Needs

Need	iPDS™ Feature
Be easy-to-use.	iPPS software and the PROM programming personality modules were designed to provide ease-of-use.
Program a wide variety of PROMs.	Personality modules each permit the programming of a family of PROMs or microcontrollers.
Be upgradeable.	New personality modules will be released as new PROM families appear.
Display (P)ROM contents in ASCII characters or in a variety of bases.	iPPS DISPLAY command displays (P)ROM (or buffer or file) contents in ASCII characters and in binary, octal, decimal, or hexadecimal.
Enable a printer to print out (P)ROM contents in ASCII characters and in a variety of bases.	iPPS PRINT command prints out (P)ROM (or file or buffer) contents in ASCII characters and in binary, octal, decimal, or hexadecimal.
Check for blank PROMs.	iPPS BLANKCHECK command checks for blank PROMS.
If PROM is not blank, check PROM contents for compatibility with program.	iPPS OVERLAY command checks PROM contents for compatibility with program.
Recognize file formats of development system object files.	iPPS command file switch allows you to indicate to the iPDS system which object file format is being used.
Support transfers of program code from development system, disks, and other PROMs to the new PROM.	iPPS COPY commands allow you to copy in either direction between the iPDS disk drive(s), PROMs, and the iPDS buffer storage.
Provide temporary storage and software for manipulating programming data before loading it into the PROM.	iPDS buffer provides temporary storage and the iPPS SUBSTITUTE and LOADDATA commands allow you to manipulate programming data before you load it into a PROM.
Load data into PROMs in a variety of formats, e.g.: —interleaving 16-bit data into two 8-bit PROMs —segmenting long programs so that resulting program segments fit into successive PROMs	iPPS FORMAT command allows you to format data in a variety of ways so that it can be loaded into PROMs in various sequences (including interleaving and segmenting).
Verify the accuracy of copying.	iPPS software automatically checks the accuracy of copying.
Compare programming buffer with PROM contents	iPPS VERIFY command compares buffer data with PROM data.
Control the security feature of advanced microcontrollers for unauthorized access.	iPPS KEYLOCK command locks advanced microcontrollers.
Automate routine PROM programming tasks.	ISIS SUBMIT files permit you to store frequently used command sequences. The files can then be activated with a single command.

APPENDIX A PROM PROGRAMMING EXAMPLES

Displaying (P)ROM contents and programming PROMs are easy tasks with the iPDS system. The following four examples show typical uses of the iPDS system's PROM programming capabilities:

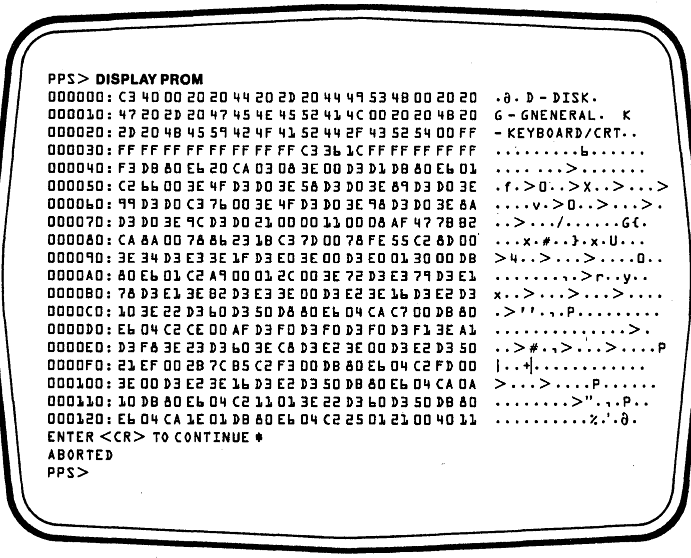
- Examining the contents of a masked ROM
- Duplicating a PROM
- Interleaving a file between two PROMs
- Locking a microcontroller

EXAMPLES

The examples assume that the iPDS system is under control of the iPPS-PDS software. The boldface characters shown on the iPDS screen displays indicate user entries. The key-in sequence below each screen display gives the actual entries that you must key in to obtain the screen display.

Examining the Contents of a Masked ROM

The DISPLAY command lets you examine the contents of a PROM or a masked ROM.

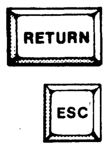


```
PPS> DISPLAY PROM
000000: C3 40 00 20 20 44 20 2D 20 44 49 53 4B 00 20 20  .∂. D - DISK.
000010: 47 20 2D 20 47 45 4E 45 52 41 4C 00 20 20 4B 20  G - GNENERAL.  K
000020: 2D 20 4B 45 59 42 4F 41 52 44 2F 43 52 54 00 FF  - KEYBOARD/CRT..
000030: FF FF FF FF FF FF FF FF C3 36 1C FF FF FF FF FF  ........6......
000040: F3 DB 80 E6 20 CA 03 08 3E 00 D3 D1 DB 80 E6 01  .... ...>.......
000050: C2 66 00 3E 4F D3 D0 3E 58 D3 D0 3E 89 D3 D0 3E  .f.>O..>X..>...>
000060: 99 D3 D0 C3 76 00 3E 4F D3 D0 3E 98 D3 D0 3E 8A  ....v.>O..>...>.
000070: D3 D0 3E 9C D3 D0 21 00 00 11 00 08 AF 47 7B 82  ..>.../......G{.
000080: CA 8A 00 78 86 23 1B C3 7D 00 78 FE 55 C2 8D 00  ...x.#..}.x.U...
000090: 3E 34 D3 E3 3E 1F D3 E0 3E 00 D3 E0 01 30 00 DB  >4..>...>....0..
0000A0: 80 E6 01 C2 A9 00 01 2C 00 3E 72 D3 E3 79 D3 E1  ........,.>r..y..
0000B0: 78 D3 E1 3E 82 D3 E3 3E 00 D3 E2 3E 16 D3 E2 D3  x..>...>...>....
0000C0: 10 3E 22 D3 60 D3 50 D8 80 E6 04 CA C7 00 DB 80  .>''.,.P.........
0000D0: E6 04 C2 CE 00 AF D3 F0 D3 F0 D3 F0 D3 F1 3E A1  ..............>.
0000E0: D3 F8 3E 23 D3 60 3E C8 D3 E2 3E 00 D3 E2 D3 50  ..>#.,>...>....P
0000F0: 21 EF 00 2B 7C B5 C2 F3 00 DB 80 E6 04 C2 FD 00  |..+|............
000100: 3E 00 D3 E2 3E 16 D3 E2 D3 50 DB 80 E6 04 CA 0A  >...>....P......
000110: 10 DB 80 E6 04 C2 11 01 3E 22 D3 60 D3 50 DB 80  ........>".,.P..
000120: E6 04 CA 1E 01 DB 80 E6 04 C2 25 01 21 00 40 11  ..........%.'.∂.
ENTER <CR> TO CONTINUE *
ABORTED
PPS>
```

280015-3

Key-In Sequence **Comments**

DISPLAY PROM



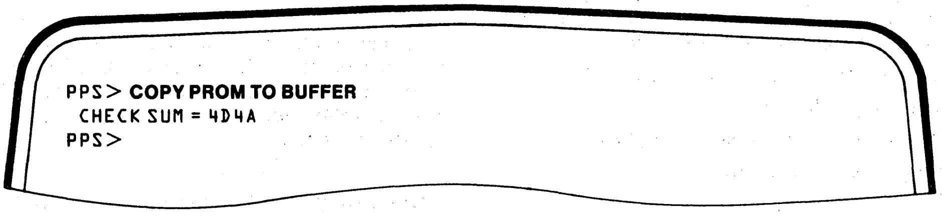
280015-4

This example shows the data in the PROM in hexadecimal format, which is the default base in this example. Press the ESC key at any time to end the display. The "S" sign is the echo of the ESC key. You can also display the data in other number bases. Note the ASCII code displayed in the far right column.

Duplicating a PROM

One frequently used application of iPDS PROM programming is copying data from a PROM into a buffer or file, then copying it into another PROM. You can perform this operation using the iPPS-PDS buffer (or an iPDS file for intermediate storage) and the iPPS-PDS COPY commands.

The following example illustrates a direct PROM-to-buffer-to-PROM duplication. If you wish to perform these examples, place the PROM in the PROM socket and reset the iPPS-PDS (using the TYPE command for your type of PROM). A 2716 EPROM that contains sample code is used in this example.



```
PPS> COPY PROM TO BUFFER
 CHECK SUM = 4D4A
PPS>
```

280015-5

Key-In Sequence

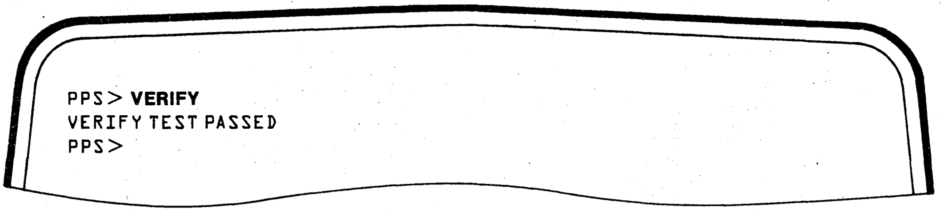
COPY PROM TO BUFFER RETURN

280015-6

Comments

This command copies every memory location in the PROM to the buffer beginning at destination address 00H in the buffer. The check-sum is the 2's complement of the 16-bit sum of all the bytes read.

If you want to check the buffer to be sure the data now there matches the original data in the PROM, one command is all that is needed. Enter the VERIFY command, and if the buffer and PROM data match, you will be informed VERIFY TEST PASSED.



```
PPS> VERIFY
VERIFY TEST PASSED
PPS>
```

280015-7

Key-In Sequence

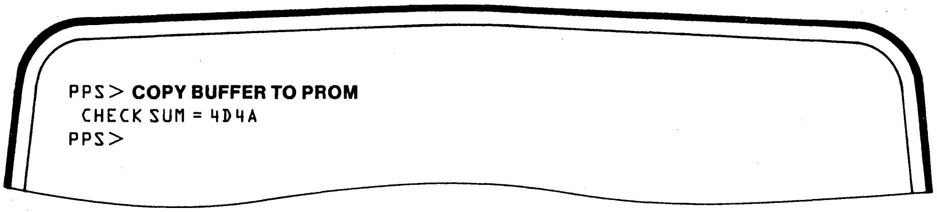
VERIFY RETURN

280015-6

Comments

The data in the buffer matches the data in the PROM.

Now that you have verified that the data in the buffer matches the data in the PROM, you are ready to copy the buffer to a blank PROM. Remove the master PROM from the PROM socket and insert the blank PROM. Then use COPY again to copy the contents of the iPPS-PDS buffer to the blank PROM.



```
PPS> COPY BUFFER TO PROM
  CHECK SUM = 4D4A
PPS>
```

280015–8

Key-In Sequence

COPY BUFFER TO PROM RETURN

280015–6

Comments

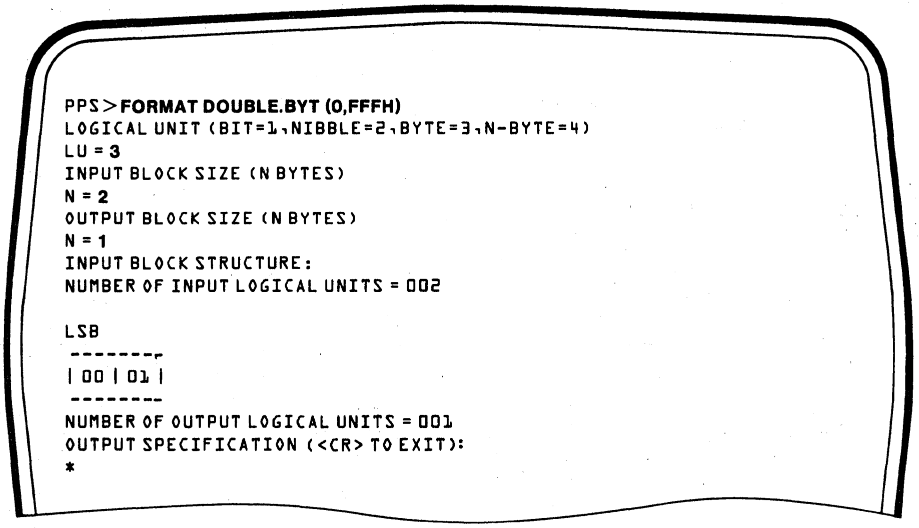
The display of the check-sum and the return of the iPPS prompt indicate that the PROM was successfully programmed.

Note that for copying from the buffer to a PROM, you do not need to use the VERIFY command. The iPPS-PDS software automatically verifies the copying when you copy in this direction.

Interleaving a File between Two PROMS

It is often desirable to have code or data arranged in 16-bit words and stored on a pair of 8-bit PROMs. This is the case, for example, when working with an 8086 microprocessor that reads from and writes to memory on a 16-bit data bus. The data is interleaved between two PROMs, the odd (or low) bytes stored in one PROM and the even (or high) bytes stored in the other PROM. The FORMAT command handles this interleaving automatically.

In the following example, a file written in Intel 8086 hexadecimal format is interleaved into two PROM devices.



```
PPS>FORMAT DOUBLE.BYT (0,FFFH)
LOGICAL UNIT (BIT=1,NIBBLE=2,BYTE=3,N-BYTE=4)
LU = 3
INPUT BLOCK SIZE (N BYTES)
N = 2
OUTPUT BLOCK SIZE (N BYTES)
N = 1
INPUT BLOCK STRUCTURE:
NUMBER OF INPUT LOGICAL UNITS = 002

LSB
--------
| 00 | 01 |
--------
NUMBER OF OUTPUT LOGICAL UNITS = 001
OUTPUT SPECIFICATION (<CR> TO EXIT):
*
```

280015-9

Key-In Sequence

FORMAT DOUBLE. BYT (0,FFFH)

RETURN

280015-6

3 RETURN

280015-6

2 RETURN

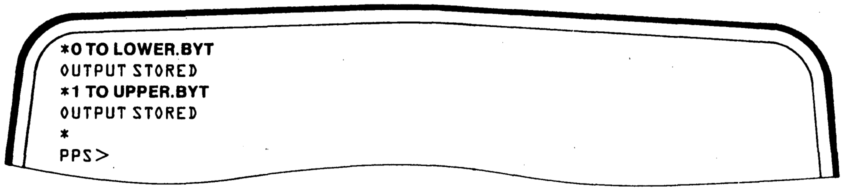
280015-6

1 RETURN

280015-6

Comments

In this example, a file called DOUBLE.BYT is split into two files, with alternate bytes being loaded into alternate files. After establishing the FORMAT command and the file name with the first entry, the iPPS software prompts for the size of the logical unit that is going to be manipulated. Byte is selected as the logical unit. You are then prompted to set up the input block size (in this case two bytes) and the output block size (one byte). A diagram of the input block is displayed with the logical units labeled. The least significant bit in the input block is displayed with the logical units labeled. The least significant bit in the input block is shown on the left. The number of logical units in the output block is also displayed. You are then prompted with an asterisk (*) to enter the output specification.



```
*0 TO LOWER.BYT
OUTPUT STORED
*1 TO UPPER.BYT
OUTPUT STORED
*
PPS>
```

280015-10

Key-In Sequence | **Comments**

0 TO LOWER.BYT

280015-6

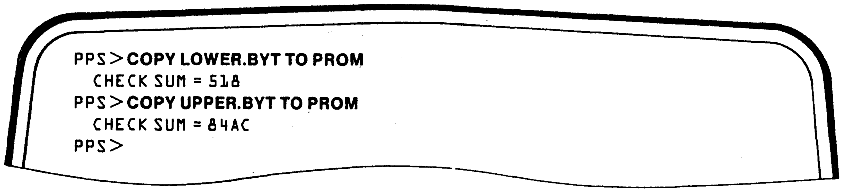
1 TO UPPER.BYT

280015-6

280015-6

Once the size of the logical unit, the input block size and the output block sizes have been established, you are prompted for the output specification (how you want the data in the file to be manipulated in terms of logical units). This example specified that the least significant byte in each input block be stored in a file titled LOWER.BYT in the default drive. The iPPS software then sorts through the DOUBLE-BYT file. Next it specifies that the most significant byte be stored in a file titled UPPER.BYT. The iPPS software then sorts through the DOUBLE-BYT file and copies every odd byte to the UPPER-BYT file. OUTPUT STORED is displayed after each output specification is implemented. You then have the option of entering another output specification. Pressing RETURN exits the FORMAT command and returns the iPPS prompt.

You can use the two files created with this FORMAT operation to program two PROMs, which you can then install in parallel to provide 16-bit data words to a 16-bit microprocessor. To copy the files the PROMs, use the COPY command as follows.



```
PPS>COPY LOWER.BYT TO PROM
  CHECK SUM = 518
PPS>COPY UPPER.BYT TO PROM
  CHECK SUM = 84AC
PPS>
```

280015-11

Key-In Sequence | **Comments**

COPY LOWER.BYT TO PROM

280015-6

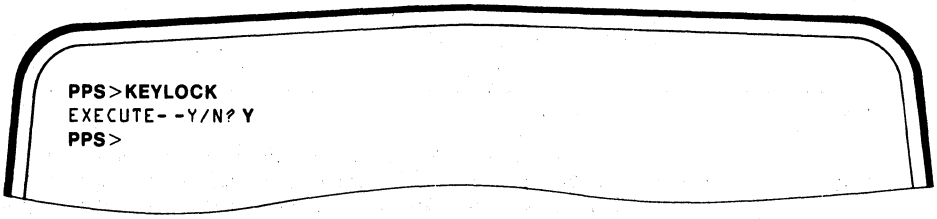
COPY UPPER.BYT TO PROM

280015-6

You must install a blank PROM in the personality module before entering each COPY command.

Locking a Microcontroller

After programming a microcontroller, you can protect it from unauthorized access by locking it with the KEYLOCK command (the KEYLOCK command cannot be used with all EPROMs). The following example locks an 8751H microcontroller, which then cannot be unlocked without erasing it.



```
PPS>KEYLOCK
EXECUTE--Y/N? Y
PPS>
```

280015–12

Key-In Sequence		Comments
KEYLOCK	RETURN 280015–6	Entering Y locks the EPROM. If you enter N, the command terminates and EPROM remains unlocked.
Y	RETURN 280015–6	

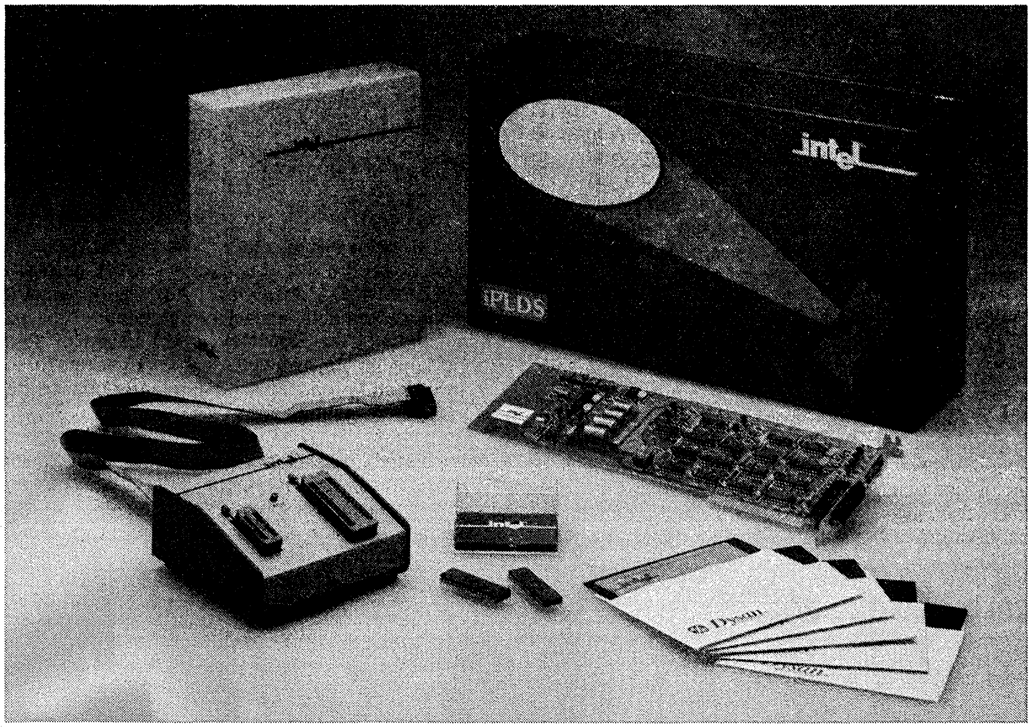
EPLD Development Tools 8

iPLDS
INTEL PROGRAMMABLE LOGIC DEVELOPMENT SYSTEM

- **Provides the Necessary Hardware and Software Tools to Quickly Turn Design Concepts Into Programmed Erasable Programmable Logic Devices (EPLD)**
- **Includes Comprehensive, Menu-driven Software with Soft Key Input and On-line Help Messages**
- **A Variety of Programmable Options Available to Program, Read, and Verify EPLD Devices**
- **Includes a Logic Optimizing Compiler That Automatically Minimizes Logic, and Produces the Best Design Fit for the Device Selected**
- **Supports a Variety of Input Methods:**
  — Schematic Capture (Optional)
  — Interactive Logic Builder
  — Boolean Equations
  — State Machine Entry (Optional)
  — Design Files Via a Text Editor
- **Generates Output in the Standard JEDEC File Format**
- **Interfaces with the IBM* PC, PC XT, PC AT, and True Compatibles**
- **Programming Tools to Obtain the Most Utilization of EPLD Resources**
- **Includes Sample Device**

OVERVIEW: The Intel Programmable Logic Development System (iPLDS) provides a powerful set of EPLD development tools. It is an easy to use hardware and software system for creating a logic design, optimizing and custom-fitting the design to a particular EPLD device, and then programming and verifying the EPLD device.

*IBM Personal Computer is a registered trademark of International Business Machines Corporation.



280168-1

October 1986
Order Number: 280168-003

FUNCTIONAL DESCRIPTION

The iPLDS simplifies using EPLD devices in circuit designs. The iPLDS provides all of the software, programming hardware, and documentation needed to convert a designer's hardware logic concept into a fully optimized, tested, and documented device. The designer accomplishes the entire process at his/her desk. The iPLDS interfaces with and runs on an IBM PC or true compatible.

The key to the ease of this process is the comprehensive set of high-level software tools, derived extensively from the techniques used in higher cost CAE workstations and software development processes. The system's software includes a wide choice of design input types, enabling designers to create and implement designs using the user interface matching their application.

As with most other programmable logic software systems, the designer can specify, test, and modify designs with advanced forms of Boolean equations. In addition, the iPLDS supports input from two powerful optional schematic capture packages (PC-CAPS* from P-CAD, and DASH-2* from Future-Net), input using the logic builder program (an easy-to-use interactive netlist entry package), optional state machine entry, and creation of an Advanced Design File (ADF) directly using the text editor, such as Intel's AEDIT text editor.

Advances have been made in each stage of the EPLD design cycle. The software compiles and optimizes the logic design, automatically determines the best way to fit the design into the EPLD device, and graphically displays the programmed device at the individual EPROM bit level. The software also programs, reads, and verifies the EPLD device using the system's programming hardware.

The software is designed for ease of use. The entire software package is comprised of nested menus. The bottom of the screen displays helpful messages, suggesting what to do. A separate help function is always available when a further explanation of a function is needed. Errors are identified with descriptive messages. The process is further simplified by the use of interactive graphics during design input and while viewing the design fit using the gate interconnect preview function.

EPLD DESIGN PROCESS

The iPLDS supports a complete design process from concept to programmed and tested components. The Intel Programmable Logic Software (iPLS) controls the entire process (refer to Figure 1).

Design Input

The logic design can be entered using any of the following methods:

Logic Builder: The logic design can easily be entered using the logic builder program, which uses a combination of questions and pictures to prompt the designer for inputs and outputs of logic elements. The program guides the designer through the entire design entry process by prompting for necessary information and showing a screen display, one device at a time, with input signals on the left side, and output signals on the right (refer to Figure 2).

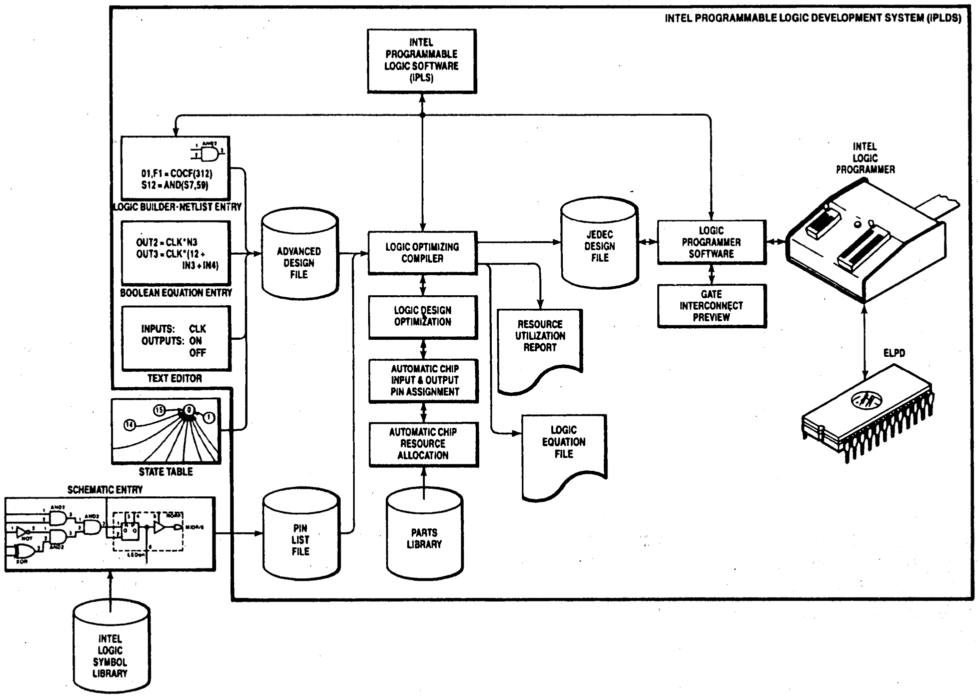
The design entry process starts with an output pin of the EPLD device. A device to drive the output pin is selected from a menu of available logic primitives. Then the system prompts the designer for the node names of each input to the primitive device. Primitive devices to drive each input node are then selected, and so on, until the entire logic circuit has been created. The circuit can be edited during initial entry and also when down-loaded from disk storage or an EPLD device. Comments can also be added. The circuit is corrected-by-design from the start, as the logic builder detects and identifies violations to basic design rules. If the design can be better described using a Boolean equation or a state machine specification, these can be directly entered into the circuit, using built-in entry functions. An Advanced Design File (ADF) is automatically created when the design input is complete.

State Machines (optional): State machine designs can be entered using the optional iSTATE software. iSTATE provides considerable flexibility in ways to specify state machine designs, supporting multiple syntaxes for state definition, specification of state transitions, inputs, and outputs and provisions for intermixing state machine and Boolean equation designs.

Once a state machine design has been coded using iSTATE, the program is input to the iPLS software which optimizes the logic, determines a best fit of the state machine design to the EPLD selected (even including automatic selection of flip-flop types), compiles the program, and produces a JEDEC or Intel hex formatted object code file for programming using one of Intel's selection of programming methods.

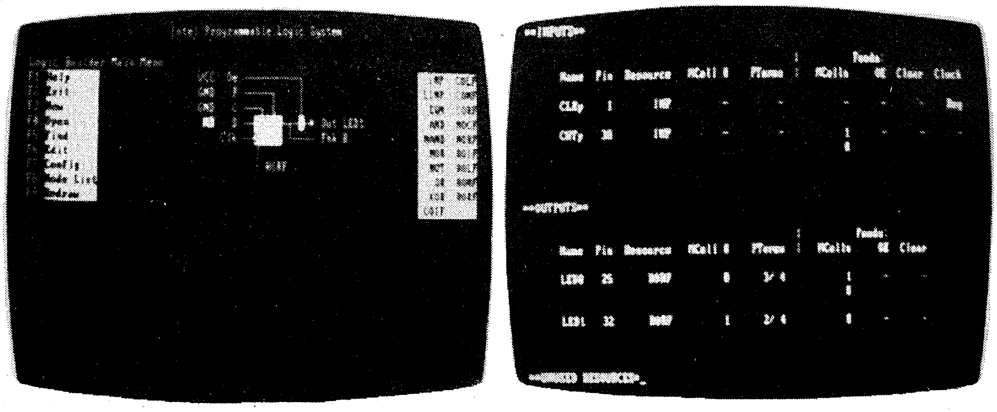
*PC-CAPS is a registered trademark of P-CAD Corporation.

*DASH-2 is a registered trademark of FutureNet Corporation.


INTEL PROGRAMMABLE LOGIC DEVELOPMENT SYSTEM (iPLDS)
INTEL PROGRAMMABLE LOGIC SOFTWARE (iPLS)
01,F1 = COCF(312)
S12 = AND(S7,59)
LOGIC BUILDER-NETLIST ENTRY
OUT2 = CLK*N3
OUT3 = CLK*(12 + IN3 + IN4)
BOOLEAN EQUATION ENTRY
INPUTS: CLK
OUTPUTS: ON OFF
TEXT EDITOR
STATE TABLE
SCHEMATIC ENTRY
ADVANCED DESIGN FILE
LOGIC OPTIMIZING COMPILER
LOGIC DESIGN OPTIMIZATION
AUTOMATIC CHIP INPUT & OUTPUT PIN ASSIGNMENT
AUTOMATIC CHIP RESOURCE ALLOCATION
PARTS LIBRARY
PIN LIST FILE
RESOURCE UTILIZATION REPORT
LOGIC EQUATION FILE
JEDEC DESIGN FILE
LOGIC PROGRAMMER SOFTWARE
GATE INTERCONNECT PREVIEW
INTEL LOGIC PROGRAMMER
ELPD
INTEL LOGIC SYMBOL LIBRARY

280168–2

Figure 1. Intel Programmable Logic Development System (iPLDS)



280168–3

Figure 2. Logic Builder and Utilization Screen Samples

Text Editor: The logic design can be entered using a text editor to create an ADF similar to most PLD design packages. The ADF provides a simple format for specifying design inputs, outputs, net lists, and Boolean logic equations.

Schematic Capture (optional): The logic design can be entered using either of two powerful schematic capture programs, PC-CAPS or DASH-2. With these schematic entry programs, specially configured to work with Intel's logic symbol libraries, logic design schematics can be drawn on the host computer's screen by interactively entering primitive logic symbols using a menu of logic symbols and a mouse. The schematic can be easily edited. Changing a design from one EPLD size to another is accomplished by changing the device number on the drawing of the schematic. Hard-copy printout and plotting of the schematic is also supported. The pin list file ouput of the schematic capture programs can then be transferred directly to the Logic Optimizing Compiler (LOC).

Boolean Equations: The logic design can be entered using Boolean equations, inserting them directly into the Logic Builder design, schematic capture design ADF (using a text editor), or state machine file.

The logic symbol libraries are available separately as iSLIBFNET and iSLIBPCAD. These products include the full device libraries to support both Intel and Altera Corp. EPLD devices, and all necessary software interfacing between the schematic capture packages and the iPLDS.

Logic Optimizing Compiler (LOC)

The LOC accepts the design input from an ADF or a pin list file. Once the logic design is accepted, the LOC begins to compile the input code in three stages.

Logic Design Optimization: The LOC converts the input file to Boolean equations. At this time, logical and syntactical error checks are made. The Boolean equations are then combined into an expanded sum-of-products form. The LOC then performs heuristically selected optimization algorithms (including De Morgan's theorem) to reduce the design to the minimum number of terms.

Automatic Chip Input and Output Pin Assignment: After the optimization is complete, the compiler automatically fits the design into the specified EPLD device using device parameters read from the software parts library. If input and output pins have not been assigned by the designer, the compiler automatically assigns them to locations that provide the best fit of the design within the EPLD device.

Automatic Chip Resource Allocation: This function determines the best possible fit of the design within the format structure of the chosen EPLD device.

Resource Utilization Report

Once the logic optimization is complete, a resource utilization report is created documenting which of a part's resources have been utilized and how the resources have been used. The report is automatically stored as a disk file. A statement at the top of the report indicates whether or not the design has been successfully implemented. This is followed by a header section detailing the designer's name, date the design was entered, EPLD device number, and the title of the design. Next is a pictorial representation of the EPLD device, with all pins labeled. Then details of the input and output pins, and any buried registers are listed. The report also lists any unused device resources, and what percentage of the device was utilized.

Logic Equation File

At the completion of the logic optimization, a Logic Equation File (LEF) is also created. The LEF is a version of the ADF with all logic minimized. The LEF shows the result of the Logic Optimizing Compiler.

JEDEC Design File

The LOC also produces a JEDEC design file of object code, which can be programmed directly into an EPLD device using the Logic Programming Software (LPS) and the Intel logic programmer. The JEDEC (Joint Electron Device Engineering Council) file format is a standard data transfer format.

JEDEC to HEX Conversion

The JEDEC file can be converted to an Intel Hex File Format using a simple conversion program. The Intel Hex File Format code can then be used to program EPLDs using Intel Universal Programmers with iUP-GUPI modules.

Logic Programmer Software (LPS)

The LPS controls the programming, reading, and verifying of the EPLD device by the Intel logic programmer. The gate interconnect preview feature of the LPS provides a windowed view into the structure of the EPLD device, graphically displaying how the design was implemented into the device (refer to Figure 3). The feature enables the designer to get a complete view of the EPLD device, showing the status of individual EPROM bits. The actual bit pattern and I/O drivers can be checked, and individual bits may be altered. This feature can be used before the EPLD device has been programmed, or after reading a previously programmed EPLD device.

Intel Logic Programmer

Programming the EPLD is accomplished by use of the Intel logic programmer, which consists of an interface card (installed in an IBM PC, or true compatible) and a separate programmer (pod) that is connected to the card by a ribbon cable. The Intel logic programmer uses a fast programming algorithm to program most designs in less than one minute.

While programming, the Intel logic programmer also performs a double verification of the bit pattern. The programmer verifies each bit after it is programmed, and it verifies the bit pattern again after the entire EPLD device is programmed.

The Intel logic programmer also programs the EPLD security and turbo bits. The security bit, once programmed, prevents programming, reading, and verifying of the device. The turbo bit, once programmed, prevents the device from going into a stand-by mode. Although the device will consume more power in turbo mode, the propagation delay through the device is reduced. To be reprogrammed, the device must be erased with ultraviolet light.

NOTE:
The iPLDS includes a programmer pod for the 300 and 1200 gate equivalent devices. Other logic programmers are available for programming devices with different pin counts and package styles.

iUP-GUPI

The iUP-GUPI is a generic module that enables the iUP-200A/201A Universal Programmer and the Intel Personal Development System (iPDS™) to accept low-cost plug in adaptors that configure the system to support a wide variety of programmable devices (including EPLDs). Table 1 lists the EPLD devices supported by the iPLDS system, and the Intel Logic Programmer pods and GUPI adaptors that will program them.

Table 1. Intel Programmer Logic Development System Programming Support

Device	Equivalent Gate Count	Intel Logic Programmer Pod	iUP-GUPI Adaptor
5C031	300	included in iPLDS	GUPI LOGIC-12
5C060	600	iLP900	GUPI LOGIC-9
5C090	900	iLP900	GUPI LOGIC-9
5C121	1200	included in iPLDS	GUPI LOGIC-12
5C180	1800	iLP1800	GUPI LOGIC-18

NOTE:
Intel Programmers also support programming of equivalent Altera Corp. second-source parts.

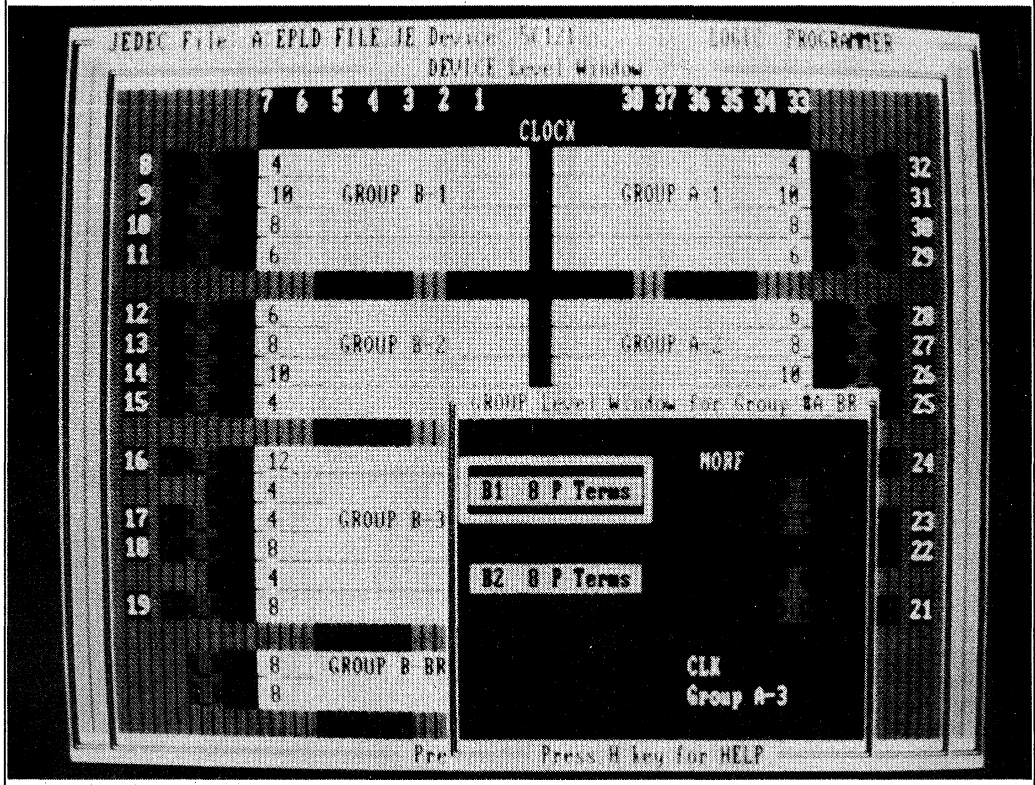
iUP-GUPI and GUPI Logic Adaptors

The iUP-GUPI and assorted GUPI LOGIC adaptors provides an alternative programming solution for Intel's H-series and Altera EPLD devices, when purchased with the iPLS, Intel's Programmable Logic Software. This complete set of software is available separately (i.e., without the iLP programmer pod and IBM interface card).

By selecting a system consisting of the iPLS software, iUP-201A (with iPPS software for the IBM PC, PC XT, or PC AT), and iUP-GUPI, no expansion slots are used in your PC (since the iUP communicates via the PC's RS232 serial port), and a more versatile programming solution is obtained. Some of the added programming advantages are stand-alone operation when several duplicate EPLDs are needed, increased device testing with checksum, verification, and optional programming of EPROMs and microcontrollers with low cost adaptors.

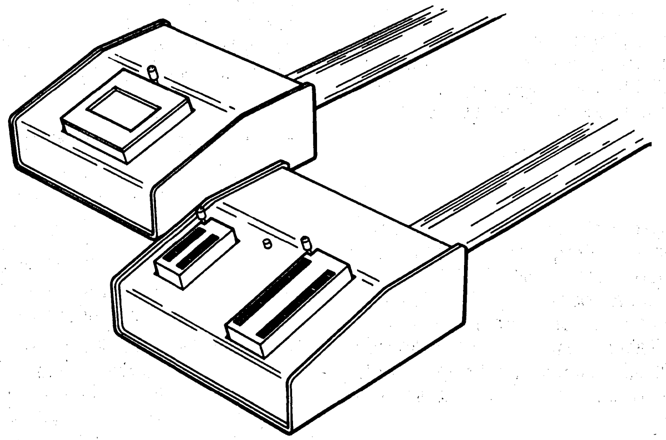
OPTIONAL PRODUCTS

iPLS:	The Intel Programmable Logic Software is available for users who do not require the logic programmer hardware. The product consists of the iPLS diskettes, sample EPLD device, and the *iPLDS User Manual* with slipcase and binder.
iUPLDSKIT09	This kit bundles all the software and hardware needed to develop EPLD's for users that already have an iUP-200A or 201A programmer. The product includes the iPLS software, iUP-GUPI module, GUPI-LOGIC09 adaptor, iPPS programming software for the IBM PC, and manuals.



280168–4

Figure 3. Gate Interconnect Preview Screen Sample



280168–5

Figure 4. Programmer Pods

Feature	Benefit
• Multiple input formats	Choice of design input methods fits the designer's background, skill level and circumstances
—Schematic capture	Widely used design methodology, provides printout of the schematic
—Logic builder	Very fast to design (no syntax errors associated with text files), easy to use, accepts Boolean equations, lower cost for some users
Boolean algebra	Traditional methodology for programmable logic
—State machine	Support for users designing with state machines
• Self contained, total system	Total, low-cost EPLD design process (development and programming)
• Logic Optimizing Compiler (LOC)	
—Automatic logic minimization and resource allocation	Optimized device utilization (allowing larger equivalent gate counts per device)
—Automatic pin assignment	Design time saved, more efficient design fit within the EPLD device
—Resource utilization report	Automatic documentation (saves time) which provides feedback on the internal implementation of the design
—JEDEC file output	Industry standard, interfaces to third party programmers
• Interactive menu driven software with help files and advanced control features for experts	Very easy to use for the novice, while the expert can quickly access the most sophisticated features
• IBM PC compatible	Minimum host computer investment, widely available, runs on MS-DOS* and PC-DOS
• Sample EPLD devices	Immediate use of the system

iSLIBFNET	The FutureNet system library provides the full device library to support both Intel and Altera Corp. EPLD devices, and all necessary software interfacing between the DASH-2 schematic capture package and iPLS. The product consists of the symbol library diskette.
iSLIBPCAD	The P-CAD system library provides the full device library to support both Intel and Altera Corp. EPLD devices, and all necessary software interfacing between the PC-CAPS schematic capture package and iPLS. The product consists of the symbol library diskette.
iLP900	The Intel Logic Programmer 900 pod is available for programming the Intel 5C060 and 5C090 logic devices (or equivalent Altera Corp. second-sourced parts).
iLP1800	The Intel Logic Programmer 1800 pod is available for programming the Intel 5C180 logic device (or an equivalent Altera Corp. second-sourced part).
iSTATE	The Intel state machine entry software package is available for entering state machine designs by specifying the state variables and state transitions.

SUMMARY

The Intel Programmable Logic Development System is a unique combination of power, versatility, and economics. It enables the logic designer to draw the schematic, check it for accuracy, compile and minimize it, program it into an EPLD device, and then revise the design and reprogram the EPLD device, all at the designer's desk.

SPECIFICATIONS

Required Hardware

The iPLDS software requires an IBM PC XT, PC AT, or other true compatible computer capable of running MS-DOS* version 2.0 or later. The computer must have a 360 KB double-sided, double-density disk drive, a hard disk, and 512 KB of RAM. Additional memory is required for the optional schematic capture programs. A color monitor is recommended, as the color graphics available provide a better representation of the data than a monochrome display.

*MS-DOS is a registered trademark of Microsoft Corporation.

The programmer interface card requires one full-size card slot in the host computer.

Operating Environment

ELECTRICAL CHARACTERISTICS

Interface card and programmer:

Static:	5V @ 300 mA ±50 mA
	12V @ 200 mA ±25 mA
Dynamic:	5V @ 300 mA ±50 mA
(programming)	12V @ 250 mA ±25 mA+device current*

NOTE:
*device current = I_{PP} + I_{CC} of the device being programmed

PHYSICAL CHARACTERISTICS

Interface card:
Width: 13.1 in. (33.7 cm)
Height: 4.2 in. (10.8 cm)

Programmer:
Width: 4.8 in. (12.3 cm)
Height: 1.9 in. (4.9 cm)
Depth: 4.8 in. (12.3 cm)

Shipping weight: 7 lbs.

ENVIRONMENTAL CHARACTERISTICS

Operating Temperature: 10°C to 40°C
Relative Humidity: 8% to 80%

Equipment Supplied

HARDWARE

- Intel Logic Programmer and cable
- Logic programmer interface card
- EPLD device

SOFTWARE

- Intel Programmable Logic Software (iPLS)-master program diskette
- Logic Builder (LB)-design entry diskette
- Logic Optimizing Compiler (LOC) diskette
- Logic Programmer Software (LPS) diskette
- Installation (INSTALL) diskette

DOCUMENTATION:

- *iPLDS User Manual,* order number 166612

ORDERING INFORMATION

Product Order Code	Description
iPLDS	Intel Programmable Logic Development System (hardware, software, sample device, and documentation)
iPLS	Intel Programmable Logic Software (software, sample device, and documentation only)
iSLIBFNET	FutureNet symbol library for use with the DASH-2 schematic capture package
iSLIBPCAD	P-CAD symbol library for use with the PC-CAPS schematic capture package
iLP900	iLP pod for programming Intel 5C060 and 5C090 logic devices
iLP1800	iLP pod for programming Intel 5C180 devices
iSTATE	iPLDS state machine entry software package

NOTE:
The DASH-2 schematic capture program is available from FutureNet Corporation, and the PC-CAPS schematic capture program is available from P-CAD Corporation.

intel®

APPLICATION NOTE

AP-279

May 1986

Implementing an EPLD Design Using Intel's Programmable Logic Development System

LAKSHMI JAYANTHI
DSO APPLICATIONS

Order Number 280310-001

OVERVIEW

Welcome to the fascinating world of ERASABLE PROGRAMMABLE LOGIC DEVICES (EPLDs) and Intel's Programmable Logic Development System (iPLDS). This application note has been written for the newcomer to Intel's devices and design tools. It has been designed as a step-by-step guide through the tools but should also prove useful as a reference document for the experienced logic designer.

By the end of this application note you will have designed/solved multiple logic problems and be in a position to implement solutions to many of the digital design challenges you face today. It is anticipated that this application note will be used in conjunction with Intel's iPLS software. To increase the usefulness of this application note, Intel will supply a PCB card for you to experiment on and a sample diskette (see Appendix E for details).

This application note is divided into the following three sections:

1. An introduction to Erasable Programmable Logic Devices (EPLD)
2. An introduction to Intel's Programmable Logic Development System (iPLDS)
3. Implementation of EPLD and iPLDS using detailed examples to implement a logic design.

INTRODUCTION

Programmable logic in the form of PALs have been available for some time. They have become more complex as Large Scale Integration (LSI) techniques have been applied to this technology.

The benefits of Large Scale Integration circuits are many fold. These circuits offer lower manufacturing costs, since the use of customized LSI circuits reduces required printed circuit board space, thereby significantly reducing board costs. These circuits also consume lower power so less expensive power supplies are required and cooling fans are also eliminated. LSI circuits also have higher reliability than equivalent systems comprised of many low density standard components.

As end users of semiconductors moved into higher and higher levels of integration, chip designers found it more and more difficult to define larger and larger blocks of logic. These difficulties led to the emergence of the user-defined Application Specific Integrated Circuit (ASIC).

The options available for application specific logic are explained below and shown in Figure 1.

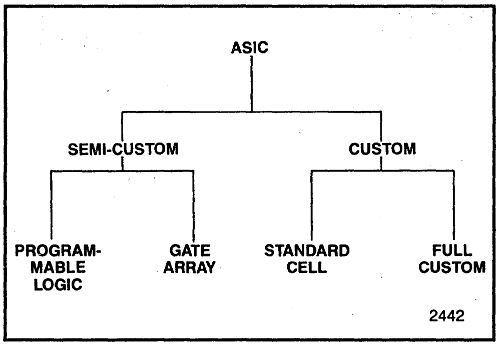


Figure 1. Logic Options

Full Custom: These circuits can be tailored to give the best functional performance with the highest level of integration, the smallest silicon area, the lowest power use, and be produced for the least cost at high production volumes.

Standard Cell Library: This approach represents an integrated circuit which is composed of predesigned and precharacterized cells chosen from a computer data base library of cells.

Gate Arrays: These are integrated circuits that contain a regular, usually square, matrix of predefined logic gates.

User Programmable Logic: The concept of user programmable logic is to provide the designer with the benefits of custom LSI chips from standard products.

A recent innovation in the programmable logic field has been Intel's introduction of an ERASABLE Programmable Logic Device. Using the same technology used in the manufacture of EPROMs, Intel now offers increased flexibility to the logic designer.

Intel has addressed the limitations of gate arrays and fuse programming logic with its EPLD products and development system support tools. The benefits to the system designer are:

- Greatly reduced lead times
- Low design costs
- Ease of design changes
- Low power dissipation from CHMOS technology
- Multiple programming facility
- Maximum flexibility in each chip and the ability to erase and reprogram
- High density products that maximize function, integration, and quality
- A self-contained, low-cost sophisticated development system based upon the industry standard IBM PC XT or AT.

Table 1. Intels EPLDs

EPLD	Gates	Pins	Dedicated Inputs	I/O
5C031	300	20	10	8
5C060	600	24	4	16
5C090	900	40	12	24
5C121	1200	40	13	24
5C180	1800	68	12	48

EPLDs are now a cost-effective solution to the problem of large scale logic integration. EPLDs are the simplest form of high density application-specific logic to implement. At present, the following logic devices are available from Intel as shown in Table 1.

Intel's EPLDs use the "Sum Of Products" architecture with programmable AND and fixed OR gates to drive a combinatorial or registered output. Each of the devices listed in Table 1 has different attributes and resources targeted at specific applications.

In general each device contains multiple sets of programmable MACROCELLS as shown in Figure 2.

Everything is programmable (and erasable if you need to make modifications). Product terms may be generated from any combination of input terms—any terms not used are considered a "don't-care" in the array. The output register is also programmable—you can choose D-type, Toggle, SR, or even JK FLIP-FLOPs; you can even choose no output register if you only require combinatorial outputs. The clock and output enables are also programmable.

Intel EPLD devices are available in many configurations to fit most applications. A complete listing of data sheet availability is covered in Appendix E.

DESIGN TECHNIQUES USING INTEL'S EPLDS

Designing with EPLDs is similar to designing with standard TTL logic circuits. The focus moves from "how can I configure this design with standard parts" to "what else could I replace using this EPLD". Remember, if you ever use all of an EPLDs resources you just move up the device chain to the next bigger component—all of the work you did is DIRECTLY PORTABLE to a larger device.

Any network, either combinatorial or registered, has an equivalent two level form. Any logic circuit consisting of AND, OR, NOR, NAND, XOR Logic can easily be converted into the corresponding truth table. Any Boolean expression, no matter how complex, may be written in Sum-Of-Products form. This Sum-Of-Products expression that has been derived from the truth table can be reduced until it has as few product terms as possible. This procedure can be repeated for any complex network.

Let us consider a very simple network as shown in Figure 3. This logic circuit consists of an AND gate, an OR gate and a NOT gate. The inputs are A, B, C, and the output is Y.

For this simple network, the truth table is shown in Table 2:

A Boolean expression can easily be written from the truth table in a Sum-Of-Products form. This expression contains the relationship between the inputs and the output.

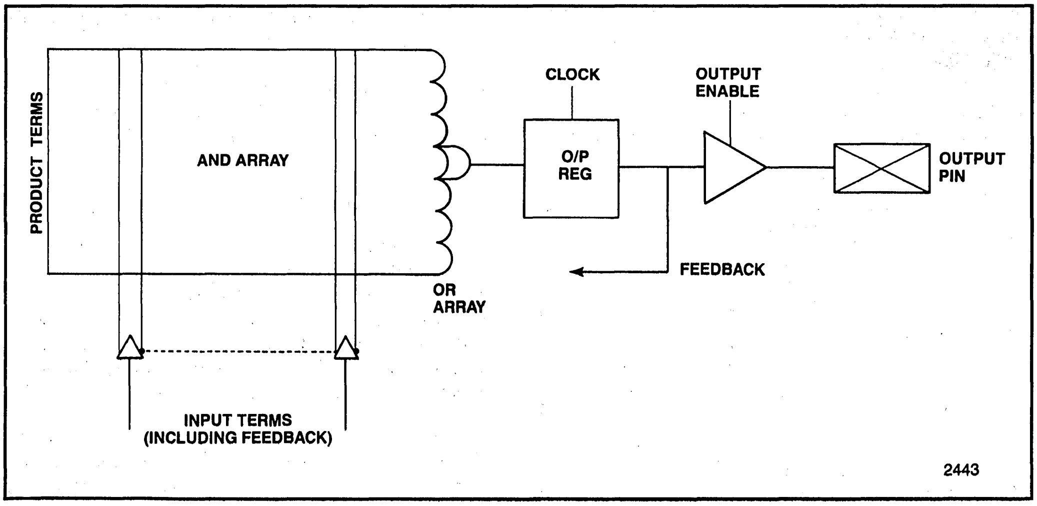


Figure 2. Macrocell Arch

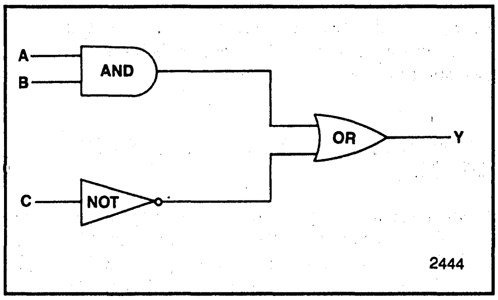


Figure 3. Simple Network

Note that the output Y is true in five of these eight states (0,2,4,6, and 7) so expressing Y in the form "Sum-Of-Products" by writing the ones in terms of A, B, and C yields:

Y = /A*/B*/C + /A*B*/C + A*/B*/C
+ A*B*/C + A*B*C

Hence, given any network, that network can be converted into its truth table. Next, a Sum-Of-Products expression that has the same truth table can be derived. If so desired, this Sum-Of-Products expression can be reduced using DeMorgan's theorem to simplify the circuit (you will see later that this will not be required).

DEVELOPMENT SUPPORT

Development tools are critical to the use of new technologies because tools allow you to control and use a new technology. Good tools help you, the designer, to work in familiar methods, then translate the design to the device.

Good tools broaden the applications by making it easy to use new technology in designs. They are not a barrier to using the technology, but encourage its use and applications.

Advanced and innovative technologies need similar advancements and innovations in the corresponding tools.

Table 2.

STATE	INPUT			OUTPUT
	A	B	C	Y
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

iPLDS, Intel's Programmable Logic Development System, provides a full spectrum of ways to design and use a variety of design tools with fast, easy-to-use entry software.

The iPLDS contains all the software, hardware, documentation and devices needed to program EPLDs. iPLDS are the most advanced PLD design tools available. It provides better utilization of device resources (more gates per chip) than any other development software. These versatile tools are for users with different skill levels and applications. iPLDS tools handle the details of converting your design to working silicon on the personal computer.

The iPLDS contains the three fundamental modules

- Logic Builder (LB)
- Logic Optimizimg Compiler (LOC)
- Logic Programmer Software (LPS)

To implement the logic design we will use the iPLDS modules in the order listed above.

The modules are essentially independant modules that use special data files to pass information as shown in Figure 4. These data files are the ADF, RPT, LEF, and JED files.

The Advanced Design File (*.ADF) is generated from the Logic Builder and contains the Inputs/outputs and all the primitive equations.

The Logic Equation File (*.LEF) contains the primitive equations that have been minimized by the Logic Optimizing Compiler.

The Utilization Report File (*.RPT) contains information on the macrocell and pin assignments.

The JEDEC File (*.JED) is the file generated by the Logic Optimizing Compiler used to program the device using the Logic Programmer.

Before implementing the logic design using the iPLDS, let us briefly discuss the iPLDS family of parts to be familiar with the iPLDS modules.

Logic Builder (LB)

The Logic Builder module guides you through the entire process of design entry by prompting for necessary information and showing a screen display (one primitive at a time) with input signals on the left side and output signals on the right side. The Logic Builder is used to generate an Advanced Design File (or ADF) by inputting the data in netlists or Boolean equations.

After all required data are entered, the Logic Builder module indicates whether the circuit is complete and properly connected. If any changes need to be made, the module enables you to edit the circuit design either by

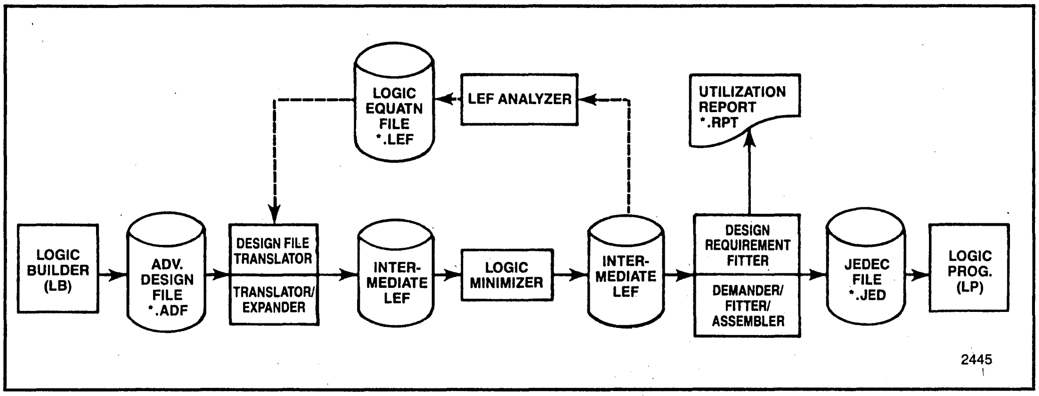


Figure 4. Block Diagram of iPLDS Modules

systematically scanning through the primitives in the Advanced Design File (ADF) or by directly finding a primitive by the name of a node connected to it.

Any circuit may be edited. The Logic Builder reads in the ADF and prompts you for changes. The Logic Builder also allows two or more partially complete ADF files to be MELDED together to form a more complex function. This concept is not discussed in this application note but will be a topic of a future application note.

Logic Optimizing Compiler (LOC)

The Logic Optimizing Compiler provides an easy-to-use interface to the Logic User System software. Regardless of the type of design entry method used, the LOC first translates an Advanced Design File (ADF) into internal logic equations; then it performs a Boolean reduction on the translated design, and finally produces a JEDEC Standard File, which is then used to program an Intel EPLD. In addition, you have the option of requesting an analysis of the Logic Equation File (LEF) as output by the Minimizer module.

The LOC performs the following functions:

- The TRANSLATOR translates the ADF into an intermediate Logic Equation File (LEF). (Most errors are detected and corrected).
- The EXPANDER expands the Boolean equations into Sum-Of-Products form, removes redundant factors from product terms, and produces another LEF.
- The MINIMIZER performs a sophisticated Boolean reduction on the translated design to maximize utilization of the EPLD.
- The LEF Analyzer converts the LEF output by the MINIMIZER into a human readable file to allow you to see your design. (*.LEF)
- The DEMANDER organizes the file output by the MINIMIZER.
- The FITTER matches your design requirements with the known resources of the Intel device.
- The ASSEMBLER converts the fitted requests into JEDEC file.

Logic Programmer Software (LPS)

The Logic Programmer Software provides a user interface to the JEDEC Standard File output of the Logic Optimizing Compiler and to the Logic Programmer Interface. You can use the Logic Programmer Software to view JEDEC files and to program your designs into EPLDs.

The Logic Programmer Software is used

- to program your designs into EPLDs
- to verify the validity of data in the device
- to read data from the device
- to display JEDEC data graphically
- to edit JEDEC data

HARDWARE REQUIREMENTS

The iPLDS requires an IBM PC XT, PC AT, or other compatible computer. A color monitor is preferred. The computer must have at least one 360K double-sided double-density disk drive, a second 360K floppy disk or hard disk, and at least 512K bytes of RAM memory.

The iPLDS consists of the Logic Programmer Interface card, and the programming unit needed to program and verify EPLDs. The Intel iUP 201 with a GUPI adapter may be used as an alternate system to program the EPLD devices.

SOFTWARE REQUIREMENTS

The personal computer should be capable of running DOS V3.0 or a higher version. The Intel Programmable Logic

Software (iPLS) that contains the software controlling the logic programmer interface and assisting in the design of Intel applications is shipped on floppy diskettes.

PROBLEM DEFINITION

We are going to use iPLDS to implement a medium complexity logic function. As a vehicle to show the usage of the tools and design techniques we will design a circuit that will roll and spin a pair of dice. The design has been split into multiple stages for illustration purposes.

This example has been chosen since it incorporates many of a typical logic design tradeoffs and also solves many of the typical problems a hardware logic designer will encounter.

Appendix A contains some basic definitions that may be useful when reading through the design and its implementation.

DESIGN SAMPLE

Problem Set-up

The circuit is designed to set both of the dice spinning when you push a switch and display a random set of numbers when you release the switch. The dice will spin at a rate that is visually pleasing and roll at the highest possible rate to ensure randomness.

You will implement the design in the following steps:

A. One dice that will roll out a number.

B. Add a switch that will control the roll/not roll action.

C. Add a second dice to roll a number.

D. Add a spinning option to both dice.

E. Retro-fit a power save feature to extend battery life.

Hence, at the end of the five design steps you will have a pair of dice spinning and showing a pair of numbers between 1 and 6 in a very random manner. At the end of the five design steps, you will have added a very realistic and practical feature to your design and that is extending the battery life by a power saving option. It is important to note that the five steps mentioned above are sequential steps in that step C can be achieved only after steps A and B etc. Let us describe the sample circuit for the dice rolling example. It is a very simple circuit allowing you to concentrate upon the design process. It illustrates the possible design stages and considerations in detail.

PART A

Four Outputs—1A, 1B, 1C, 1D are required to drive the LEDs arranged in a DICE pattern as shown in Figure 5.

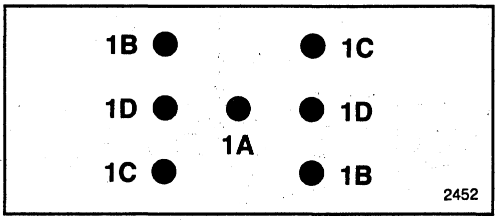


Figure 5. Dice Configuration

Operating sequence—Rolling dice from 1 to 6 and the block diagram of the circuit, both shown in Figure 6.

The total number of states that are possible is 16 since the four LED pairs generate a permutation of (2**4) = 16. The LEDs should be lit up such that any number between 1 and 6 inclusive is shown. Hence, out of the 16 possible states, only six states are valid. This leaves ten invalid states.

If the LEDs come up in a valid state upon power up, then a number between 1 and 6 will be displayed.

However, if the LEDs come up in an invalid state upon power up, then you have to design the circuit such that any one of the ten invalid states will fall into a valid state.

If the LEDs fall into any one of the ten invalid states, then you have designed the circuit to move into a state where 1A, 1B, 1C, 1D have zero logic values respectively on the next clock edge. Every time a zero logic value appears in the invalid states, then at the next clock edge, LED 1A gets lit up generating a valid state. Since 1 is a valid state, the numbers between 1 and 6 inclusive will be displayed at all subsequent clock edges.

Listed below are the steps involved in designing the logic circuit.

STEP 1. Generate the state diagram to clearly show the operating sequence including the status of the outputs for each state and the influence of the inputs on the next state transitions as shown in Figure 7. We have arbitrarily chosen that the states should count 1,2,3,4,5,6, and repeat. You could have implemented the design using any sequence but we chose the most obvious. Note how most of the invalid states move you to state 0 which then puts us into a valid state which then repeats forever.

STEP 2. Generate a truth table with entries for all available states and combinations of inputs, and use the next states resulting from these as shown in Table 3. The bracketed numbers, (3) etc., show the number being

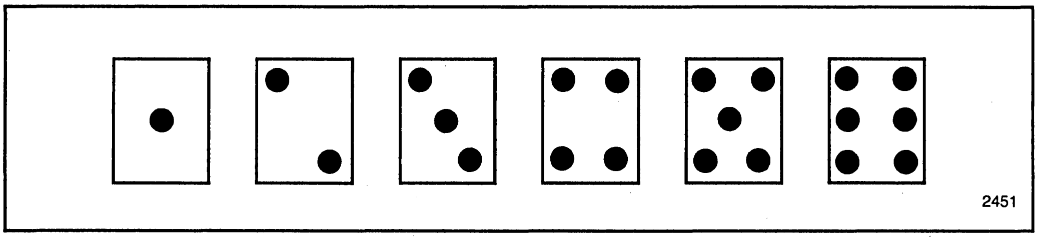


Figure 6. Rolling Sequence

displayed on the dice and the 0, 1 values of 1D, 1C, 1B, and 1A indicate which LEDs should be OFF/ON to display the required dice pattern.

STEP 3. Convert the truth table directly into Sum-Of-Products equations as shown below:

DICE1A has four entries; 3 from the valid states and one to control the invalid states

DICE1A = (/1A*1B*/1C*/1D + /1A*1B*/1C*/1D
+ /1A*1B*1C*1D + /1A*/1B*/1C*/1D)

DICE1B has five entries from valid states

DICE1B = (1A*/1B*/1C*/1D + /1A*1B*/1C*/1D
+ 1A*1B*/1C*/1D + /1A*1B*1C*/1D
+ 1A*1B*1C*/1D)

DICE1C has three entries from valid states

DICE1C = (1A*1B*/1C*/1D + /1A*1B*1C*/1D
+ 1A*1B*1C*/1D)

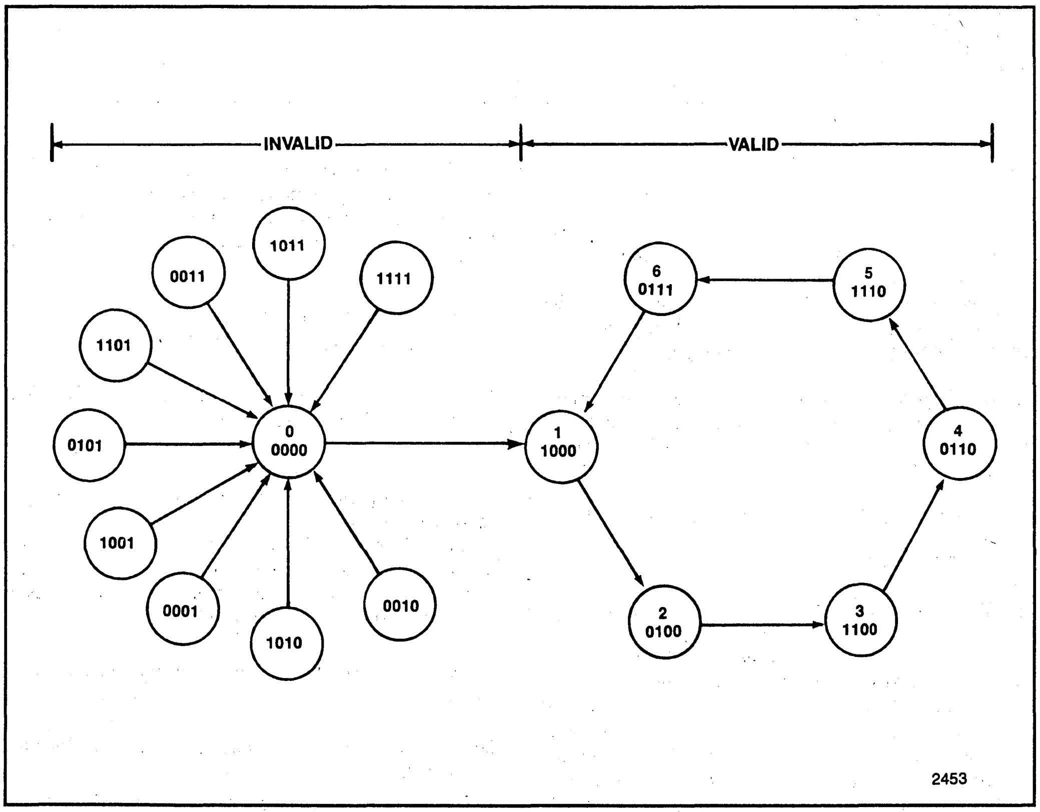


Figure 7.

Table 3. Truth Table for DICE1

Input State				Output State							
1A	1B	1C	1D	1A	1B	1C	1D	1A	1B	1C	1D
				Valid state				Invalid state			
				CHANGE TO THE NEXT VALID STATE							
1	0	0	0(1)	0	1	0	0(2)				
0	1	0	0(2)	1	1	0	0(3)				
1	1	0	0(3)	0	1	1	0(4)				
0	1	1	0(4)	1	1	1	0(5)				
1	1	1	0(5)	0	1	1	1(6)				
0	1	1	1(6)	1	0	0	0(1)				
				CONTROL THE INVALID STATES							
0	0	1	0					0	0	0	0
1	0	1	0					0	0	0	0
0	0	0	1					0	0	0	0
1	0	0	1					0	0	0	0
0	1	0	1					0	0	0	0
1	1	0	1					0	0	0	0
0	0	1	1					0	0	0	0
1	0	1	1					0	0	0	0
1	1	1	1					0	0	0	0
0	0	0	0					1	0	0	0(1)

DICE1D has one valid entry

DICE1D = (/1A*1B*1C*1D)

Note that no attempt has been made to minimize these equations - the iPLS software that you will use later contains reducing algorithms and other techniques to optimize the design. This allows you to focus upon the problem and not on tasks such as Karnaugh map reduction which a computer can often do better anyway.

Having designed part A of the circuit, you can now move on to tool usage to implement the design. Refer to the Intel Programmable Logic Software Manual if you have not installed the iPLS software.

In order to invoke iPLS type the following command

```
C:\IPLS>IPLS <Enter>
```

The iPLS menu will appear as shown in Screen 1.

The number to the left of each function allows you to select a function with a function key. Two kinds of function keys are available: toggle keys and field keys. <F3> and <F4> are toggle keys. All other keys are field keys. Functions beyond <F10> are executed by pressing the <Shift> key together with the function key. Press <F3> to invoke the Logic Builder and observe the Logic Builder menu as shown in Screen 2.

The first prompt asks for the file name. If the file already exists, its header information and primary inputs and outputs are displayed. If you enter a new file name, the Logic Builder module prompts for all the functions remaining on the screen.

```
Enter: DICE1 <Enter>
Create New Netlist(Y/N):Y
```

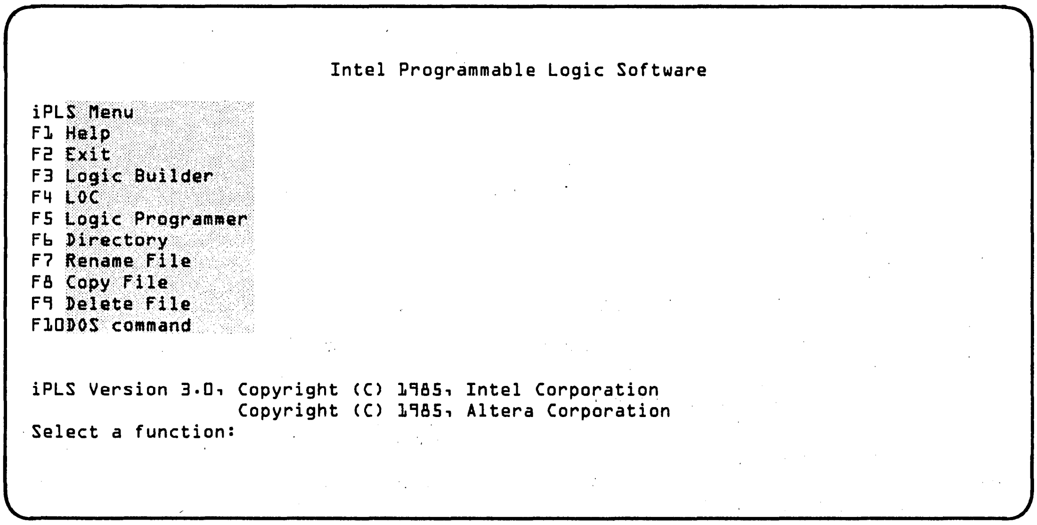
In this sample session, user entries are all in uppercase letters. Note: IPLS is case sensitive.

When initially invoked, the Logic Builder module displays its configuration menu. The Logic Builder configuration menu shows "5C121" as the default Intel part and, on the right side of the menu, displays those primitives that are legal for use with the 5C121. As soon as you enter another part (e.g. 5C060) the list of primitives changes to display the primitives applicable to that specific part.

Press <F6> and enter 5C060 when prompted for user entry.

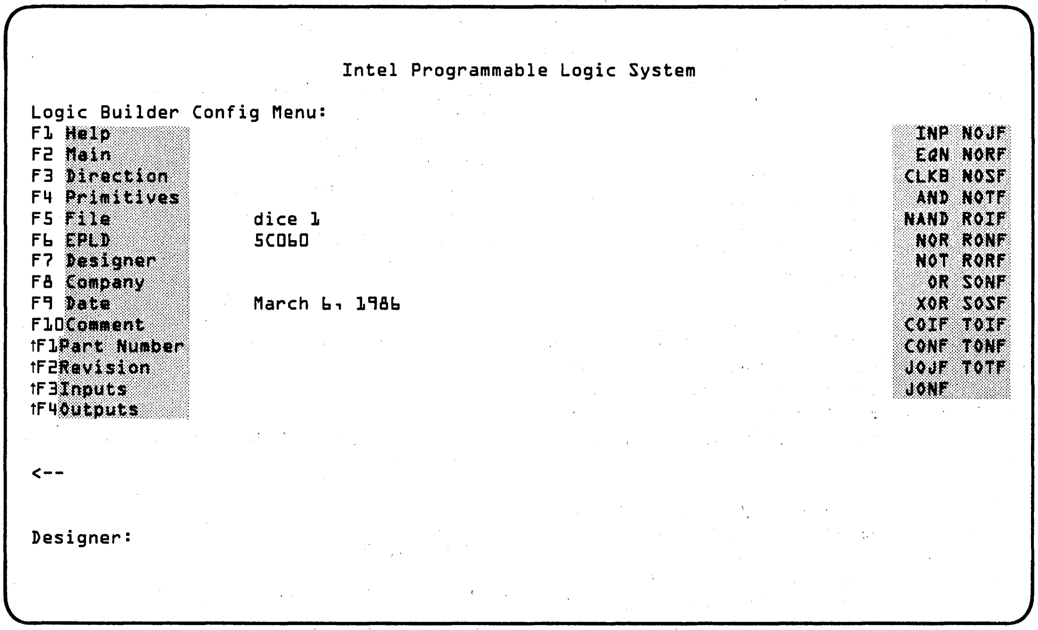
Screen 2 shows the Logic Builder Configuration Menu for the 5C060.

- The left side of the screen shows a menu of functions, each preceded by a function key number.



```
                         Intel Programmable Logic Software

iPLS Menu
F1 Help
F2 Exit
F3 Logic Builder
F4 LOC
F5 Logic Programmer
F6 Directory
F7 Rename File
F8 Copy File
F9 Delete File
F10DOS command


iPLS Version 3.0, Copyright (C) 1985, Intel Corporation
                  Copyright (C) 1985, Altera Corporation
Select a function:
```

Screen 1.



```
                          Intel Programmable Logic System

Logic Builder Config Menu:
F1 Help                                                               INP NOJF
F2 Main                                                               EQN NORF
F3 Direction                                                         CLKB NOSF
F4 Primitives                                                         AND NOTF
F5 File              dice 1                                          NAND ROIF
F6 EPLD              5C060                                            NOR RONF
F7 Designer                                                           NOT RORF
F8 Company                                                             OR SONF
F9 Date              March 6, 1986                                    XOR SOSF
F10Comment                                                           COIF TOIF
↑F1Part Number                                                       CONF TONF
↑F2Revision                                                          JOJF TOTF
↑F3Inputs                                                            JONF
↑F4Outputs


<--


Designer:
```

Screen 2.

Table 4.

Prompt	User Entry
F6 EPLD	5C060
F7 Designer	Your Name
F8 Company	Your Company
F9 Date	Present Date
F10 Comment	Our first design
↑F1 Part Number	0.1
↑F2 Revision	1.0
↑F3 Inputs	CLOCK@1
↑F4 Outputs	DICE1A@10,DICE1B@9,DICE1C@8,DICE1D@7

- The right side of the screen shows the list of available primitives (these are discussed in detail later).
- The two lines at the bottom of the screen are designated for comments (first line) and prompts (second line).
- The center of the screen is used to show a representation of the primitive; name and pictorial representation are in the middle, input signals are to the left, and output signals are to the right of the primitive.
- The direction of the arrow located on the left side of the screen below the list of functions determines the starting point and direction of design entry. If the arrow points to the left, entry is from output pins to input pins. If the arrow points to the right, entry is from input pins to output pins.

NOTE

We have assigned pin numbers to pin names by using the "@" symbol within the name of the logic variable. Specific pin numbers need not be assigned if not desired. In that case, the Logic Builder will assign its pin numbers for you.

Type in the information as given in Table 4 in the Logic Builder Config Menu. The information is also shown in Screen 3. After entering all of this required information, iPLDS will automatically prompt you through defining the circuit, starting with a primitive to drive the last output specified.

Once in the Logic Builder main menu, you are guided with prompts to enter information as follows:

Enter the name of the primitive to connect to the first node. The name may be entered by typing the name of the primitive, which highlights the appropriate primitive on the right side of the menu, then pressing <Enter>.

Subsequently, a representation of the primitive is displayed in the center of the screen surrounded by input and output signals. You are prompted for names of nodes to connect to each of the signals. The Design Primitives library contains approximately 80 basic functional blocks needed for designing circuits in programmable logic products.

Design Primitives are divided into the following groups:

- Input Primitives (INP,LINP)
- Logic Primitives (AND,GND,CLKB,NOT,VCC,OR,NAND,NOR,XOR)
- Equation Primitives (EQN)
- I/O Primitives (JOJF, NOJF, NORF, RORF, etc)

Refer to Appendix A for an explanation of the Primitives used in this example.

The logic is based on input clock transitions. At the rising edge of the clock we want the LEDs to generate a particular state depending on the input state. You want the output of the LEDs to follow the input, which is basically a D-TYPE FLIP-FLOP. You also require the feedback to generate the next state, which means that you should use a D-TYPE FLIP-FLOP with FEEDBACK or RORF as shown in Screen 4.

NOTE

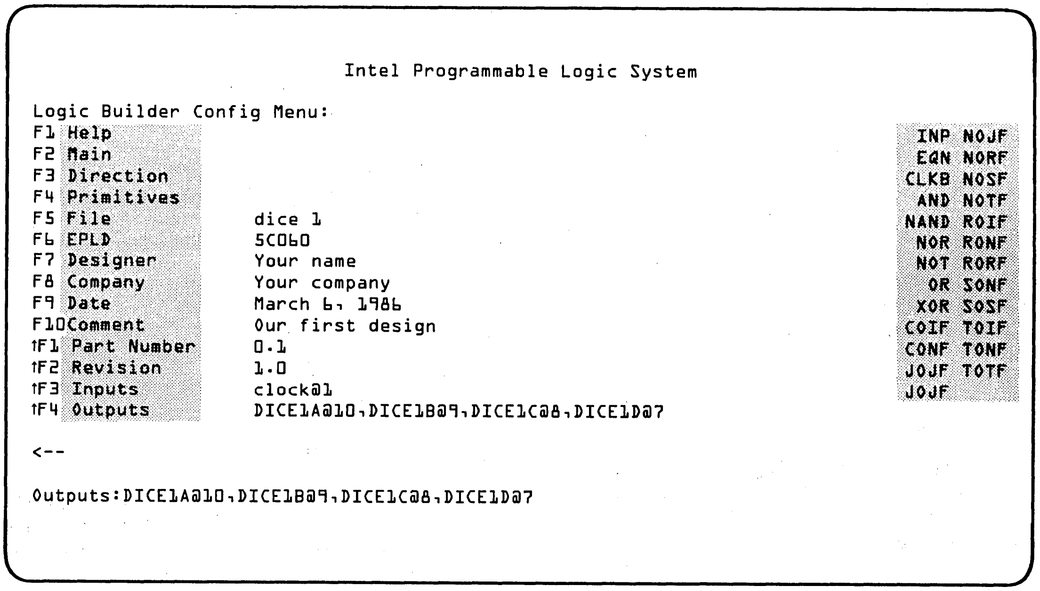
The Logic Builder module starts with the last output entered.

When you are prompted to select a primitive to drive DICE1D enter:

```
Select a primitive to drive DICE1D@7:
RORF <Enter>
```

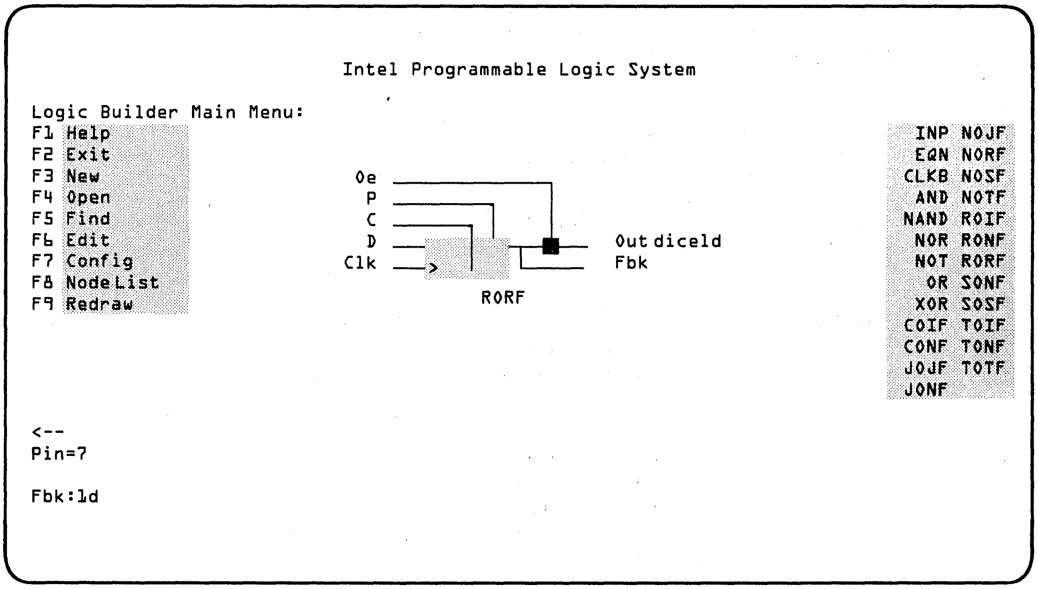
Now you are prompted for the remaining connections:

```
For FBK: 1D <Enter>

For OE, P, C: Press <Enter> (VCC, GND are
the defaults).

For D: IN1D <Enter>

For CLK: CLOCK <Enter>

Select a primitive to drive CLOCK: INP
<Enter>
```



```
                    Intel Programmable Logic System

Logic Builder Config Menu:
F1 Help                                                   INP  NOJF
F2 Main                                                   EQN  NORF
F3 Direction                                             CLKB  NOSF
F4 Primitives                                             AND  NOTF
F5 File          dice 1                                  NAND  ROIF
F6 EPLD          5C060                                    NOR  RONF
F7 Designer      Your name                                NOT  RORF
F8 Company       Your company                              OR  SONF
F9 Date          March 6, 1986                            XOR  SOSF
F10Comment       Our first design                        COIF  TOIF
↑F1 Part Number  0.1                                     CONF  TONF
↑F2 Revision     1.0                                     JOJF  TOTF
↑F3 Inputs       clock@1                                 JOJF
↑F4 Outputs      DICE1A@10,DICE1B@9,DICE1C@8,DICE1D@7

<--

Outputs:DICE1A@10,DICE1B@9,DICE1C@8,DICE1D@7
```

Screen 3.



```
                    Intel Programmable Logic System

Logic Builder Main Menu:
F1 Help                                                   INP  NOJF
F2 Exit                                                   EQN  NORF
F3 New                                                   CLKB  NOSF
F4 Open                                                   AND  NOTF
F5 Find                                                  NAND  ROIF
F6 Edit                                                   NOR  RONF
F7 Config                                                 NOT  RORF
F8 Node List                                               OR  SONF
F9 Redraw                                                 XOR  SOSF
                                                         COIF  TOIF
                                                         CONF  TONF
                                                         JOJF  TOTF
                                                         JONF
<--
Pin=7

Fbk:1d
```

Screen 4.

```
In: CLOCK <Enter>

Select a primitive to drive IN1D: EQN
<Enter>
```

After you are prompted for the equation, type it in as derived in the Problem Set-up section. Please note that "/" indicates a logical "NOT", "*" indicates a logical "AND", and " + " indicates a logical "OR". The equation is terminated by a ";" as shown in Screen 5.

```
IN1D = (1A * 1B * 1C * /1D); <Enter>
```

The following prompts and design entries, as shown in Table 5, are needed to complete the design entries for DICE1C, DICE1B, and DICE1A respectively.

The Logic Builder will stop prompting for primitives once you have entered the complete design.

Press <F8> to show the design so far as shown in Screen 6.

Press <F2> to exit.

The Logic Builder main menu is cleared, replaced by the Logic Builder exit menu.

To save the configuration and return to iPLS menu you must press <F6> (Save-Exit).

Note that you are saving the Advanced Design File (ADF) that is generated by the Logic Builder.

You can print the ADF file that has been created at the end of this session if you so desire. You can use <F10> when in the iPLS main menu to print the ADF file for a listing. You can verify your file with the DICE1.ADF file given in Appendix D. If you desire a listing, while you are in the iPLS main menu, type the following:

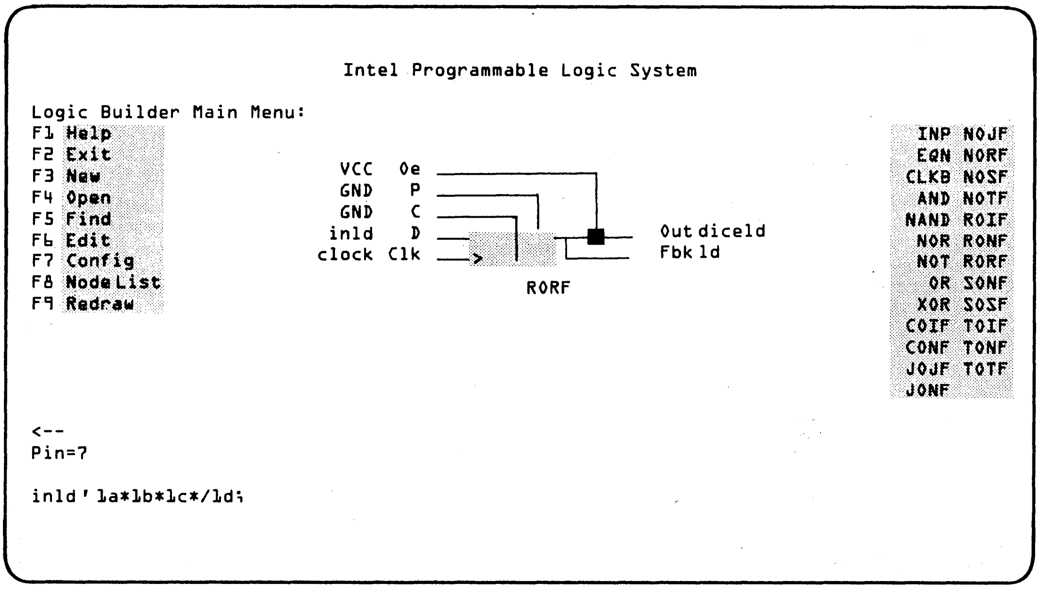
```
<F10> <Enter>

PRINT DICE1.ADF <Enter>
```

Submitting the ADF to the LOC

This ADF file is now compiled using the Logic Optimizing Compiler. To enter the ADF created with the Logic Builder module into the Logic Optimizing Compiler (LOC), press <F4> to access the LOC menu.

Table 5.

PROMPT	USER ENTRY
Select a primitive to drive 1C:	RORF <Enter>
Out:	DICE1C <Enter>
Oe:	VCC<Enter>
P:	GND <Enter>
C:	GND <Enter>
D:	IN1C <Enter>
Select a primitive to drive IN1C:	EQN <Enter>
IN1C:	(1A*1B*/1C*/1D)+(/1A*1B*1C*/1D)+(1A*1B*1C*/1D); <Enter>
Select a primitive to drive 1B:	RORF <Enter>
Out:	DICE1B <Enter>
Oe:	VCC <Enter>
P:	GND <Enter>
C:	GND <Enter>
D:	IN1B <Enter>
Select a primitive to drive IN1B:	EQN <Enter>
IN1B:	(1A*/1B*/1C*/1D)+(/1A*1B*/1C*/1D)+(1A*1B*/1C*/1D) +(/1A*1B*1C*/1D)+(1A*1B*1C*/1D);<Enter>
Select a primitive to drive 1A:	RORF <Enter>
Out:	DICE1A <Enter>
Oe:	VCC <Enter>
P:	GND <Enter>
C:	GND <Enter>
D:	IN1A <Enter>
Select a primitive to drive IN1A:	EQN <Enter>
IN1A:	(/1A*1B*/1C*/1D)+(/1A*1B*1C*/1D)+(/1A*1B*1C*1D) +(/1A*/1B*/1C*/1D);<Enter>



```
                    Intel Programmable Logic System

Logic Builder Main Menu:
F1 Help                                                      INP NOJF
F2 Exit                                                      EQN NORF
F3 New         VCC   Oe                                     CLKB NOSF
F4 Open        GND    P                                      AND NOTF
F5 Find        GND    C                                     NAND ROIF
F6 Edit        inld   D                  Out diceld          NOR RONF
F7 Config      clock Clk                 Fbk 1d              NOT RORF
F8 Node List                                                  OR SONF
F9 Redraw                  RORF                              XOR SOSF
                                                            COIF TOIF
                                                            CONF TONF
                                                            JOJF TOTF
                                                            JONF
<--
Pin=7

inld ' 1a*1b*1c*/1d;
```

Screen 5.

Once the LOC menu is displayed, you are prompted through the LOC menu functions as follows:

The Input Format prompts you to specify your form of input: If input is in the form of a pinlist as output by DASH-2, enter P, if input is an Advanced Design File, enter an ADF or press <Enter> (ADF is the default). If output is a component list from PCAD, enter C.

```
INPUT FORMAT: A <Enter>

FILE NAME: DICE1 <Enter>

MINIMIZATION: <Enter to select default>

INVERSION CONTROL: <Enter to select
default>

LEF ANALYSIS: <Enter to select default>
```

After you have answered all the prompts, you are asked if you wish to run under the above conditions as shown in Screen 7.

```
DO YOU WISH TO RUN UNDER THE ABOVE CONDI-
TIONS [Y/N]?
```

Enter: Y

Finally you are prompted with:

```
WOULD YOU LIKE TO IMPLEMENT ANOTHER DE-
SIGN [Y/N]?
```

Enter: N

Note that the LOC generates a synopsis of its progress as shown in Screen 8. You are returned to the iPLS menu.

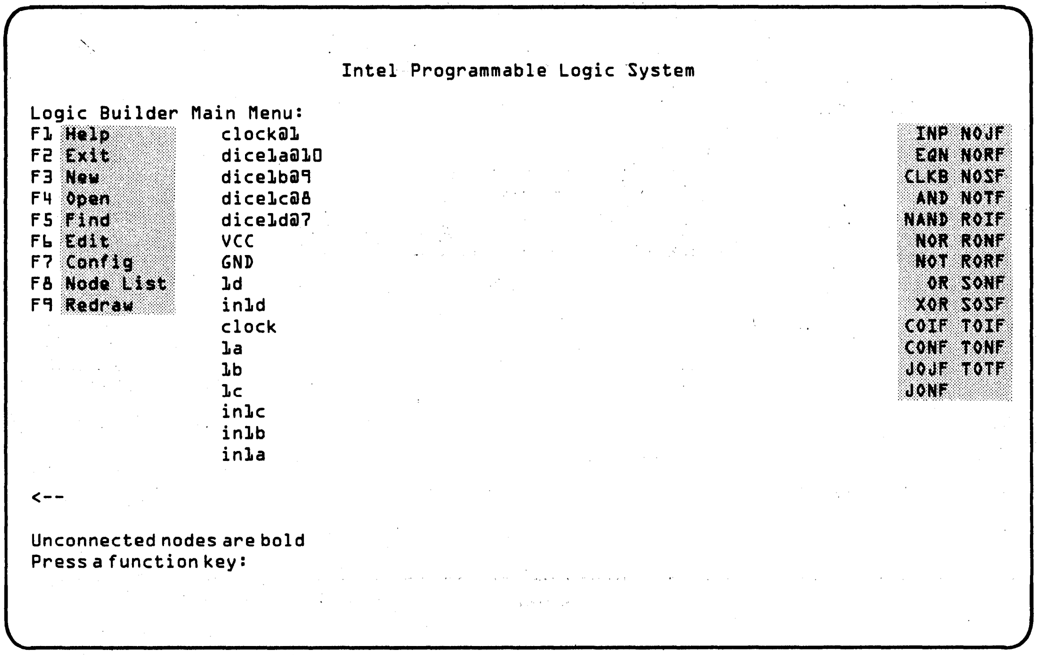
At the end of the LOC a JEDEC Standard File has been created which we will use in the Logic Programmer, DICE1.JED.

Also at the end of the LOC a report file is created, DICE1.RPT, which gives the pin configuration menu of the device. The DICE1.RPT file is given in Appendix D.

Programming the EPLD

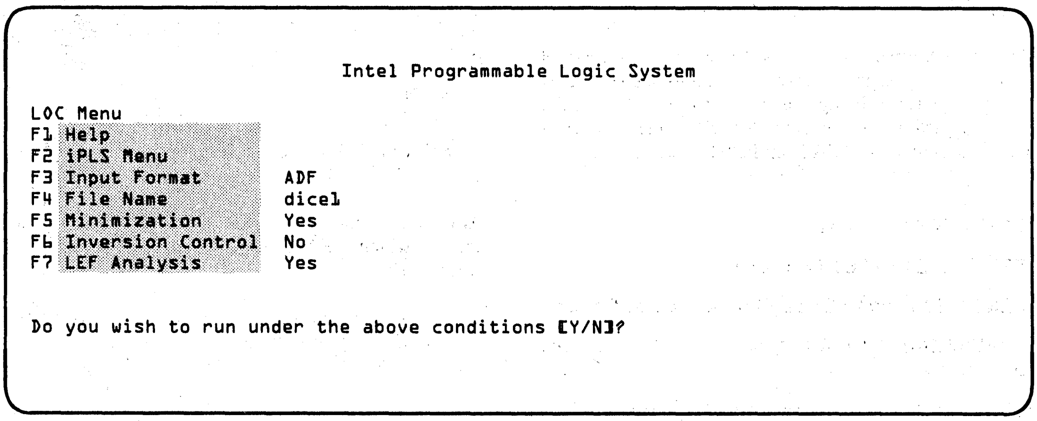
Finally, you submit your design to the Logic Programmer. In order for you to use the Logic Programmer, you must have the programming card plugged in. Please refer to the Intel Programmable Logic Software User Manual for installation instructions.

Alternatively you can use Intel's GUPI (Generic Universal Programmer Interface) to program your device.



```
                              Intel Programmable Logic System

Logic Builder Main Menu:
F1 Help          clock@1                                                         INP  NOJF
F2 Exit          dice1a@10                                                       EQN  NORF
F3 New           dice1b@9                                                       CLKB  NOSF
F4 Open          dice1c@8                                                        AND  NOTF
F5 Find          dice1d@7                                                       NAND  ROIF
F6 Edit          VCC                                                             NOR  RONF
F7 Config        GND                                                             NOT  RORF
F8 Node List     1d                                                               OR  SONF
F9 Redraw        in1d                                                            XOR  SOSF
                 clock                                                          COIF  TOIF
                 1a                                                             CONF  TONF
                 1b                                                             JOJF  TOTF
                 1c                                                             JONF
                 in1c
                 in1b
                 in1a

<--

Unconnected nodes are bold
Press a function key:
```

Screen 6.

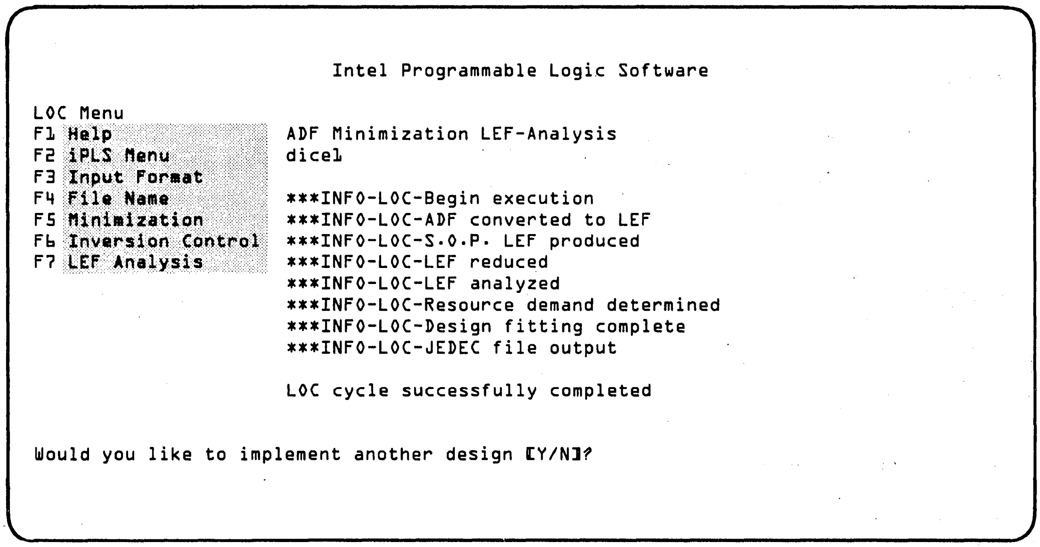


```
                              Intel Programmable Logic System

LOC Menu
F1 Help
F2 iPLS Menu
F3 Input Format          ADF
F4 File Name             dice1
F5 Minimization          Yes
F6 Inversion Control     No
F7 LEF Analysis          Yes

Do you wish to run under the above conditions [Y/N]?
```

Screen 7.

The iUP-GUPI and assorted GUPI LOGIC adaptors provide an alternative programming solution for Intel's H-series and EPLD devices, when purchased with the iPLS. This complete set of software is available without the Logic Programmer pod and the IBM interface card.

While you are still in the iPLS menu, press <F5>. This function allows you to access the Logic Programmer Software. The Logic Programmer will now come up as shown in Screen 9.



```
                              Intel Programmable Logic Software

LOC Menu
F1 Help                   ADF Minimization LEF-Analysis
F2 iPLS Menu              dice1
F3 Input Format
F4 File Name              ***INFO-LOC-Begin execution
F5 Minimization           ***INFO-LOC-ADF converted to LEF
F6 Inversion Control      ***INFO-LOC-S.O.P. LEF produced
F7 LEF Analysis           ***INFO-LOC-LEF reduced
                          ***INFO-LOC-LEF analyzed
                          ***INFO-LOC-Resource demand determined
                          ***INFO-LOC-Design fitting complete
                          ***INFO-LOC-JEDEC file output

                          LOC cycle successfully completed

Would you like to implement another design [Y/N]?
```

Screen 8.

Use the cursor keys to select "Program Device" option.

When you are prompted

Enter JEDEC file name

Enter: `DICE1.JED <Enter>`

When you are prompted for:

Select Device For Programming

Enter: `5C060 <Enter>`

When you are prompted for:

Do you wish to enable verify protection? [Y/N]?

Enter: N

When you are prompted for:

Do you wish to enable turbo-bit? [Y/N]?

Enter: N

Once you have answered all the prompts, the device is programmed and ready to be used in an actual circuit, as shown in Screen 10.

Exit from the Logic Programmer after saving the JEDEC file by using the "EXIT" option.

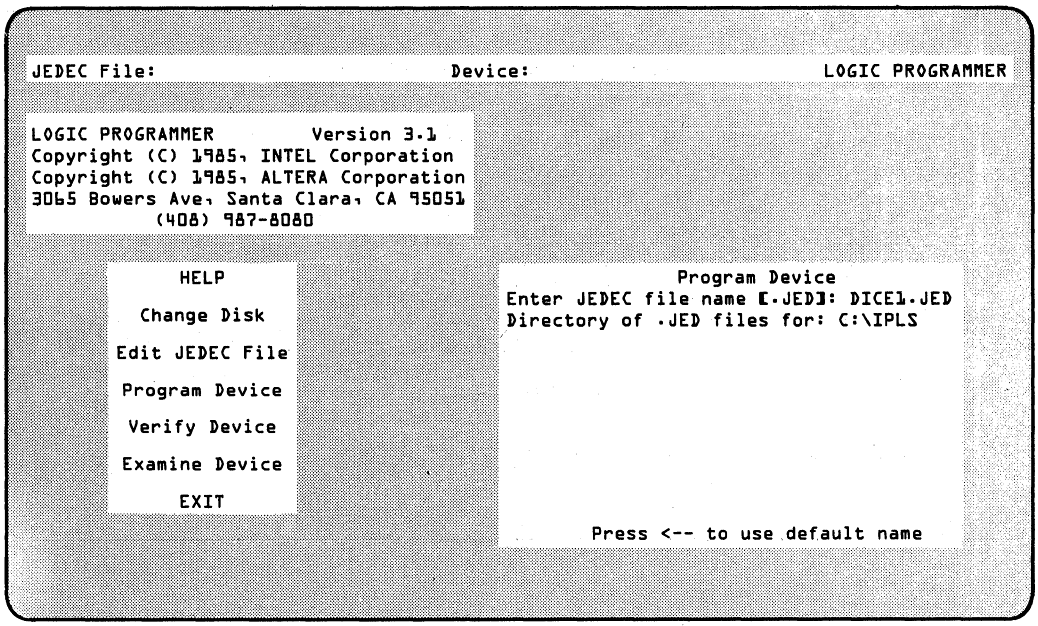
This completes part A of the design, which was to roll a single dice. The programmed device can be tested as described in Appendix C.

PART B

Now that you have a good understanding of the manner in which a circuit is designed and also a good understanding of how the programming tools are used to program the device, you can proceed to the next step in the five stages of the dice design. According to the truth table generated in part A, the dice will roll a number between 1 and 6 inclusive as long as you supply a power source. When you disconnect the power source, all the LEDs will turn off. This will not be much help since you can only see the dice roll, but not actually see a number displayed.

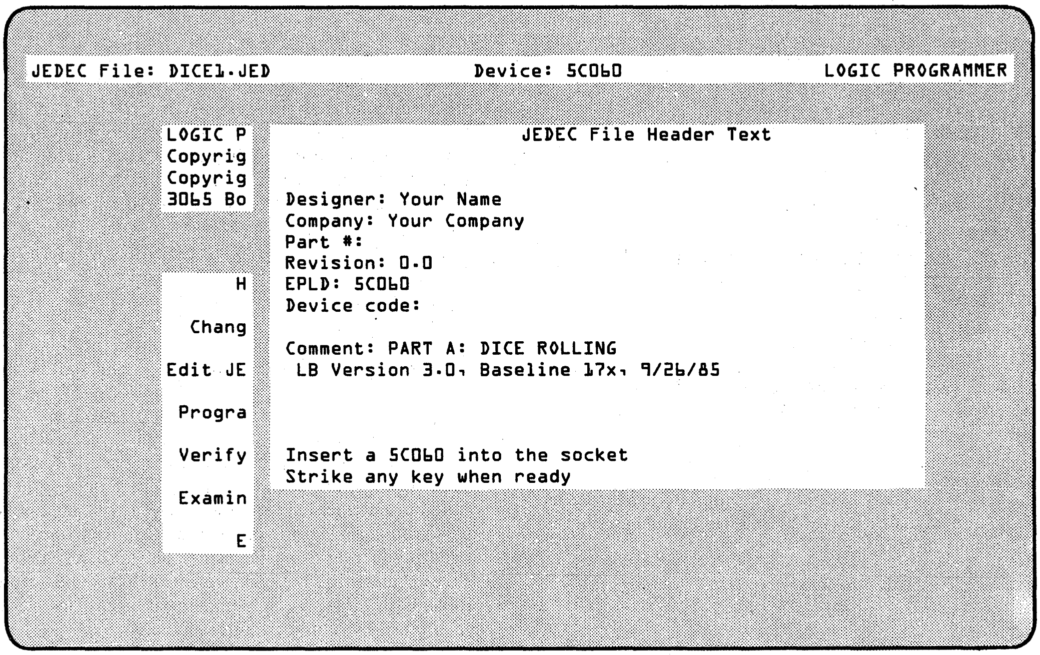
Let us include an additional feature into the rolling dice. Let us include a switch to control the rolling and display of the dice.

You could choose to gate the clock of the dice or add the necessary inputs to the product terms to effect this design. If you were to stop after this step, then gating the clock would be a simpler choice, however, you will require the dice to roll during part D of the design; so we will choose to add product terms at this stage. This also results in a better engineering solution since gated clocks often cause problems in large systems, and it has been shown that synchronous systems are more reliable.



```
JEDEC File:                         Device:                    LOGIC PROGRAMMER

LOGIC PROGRAMMER        Version 3.1
Copyright (C) 1985, INTEL Corporation
Copyright (C) 1985, ALTERA Corporation
3065 Bowers Ave, Santa Clara, CA 95051
          (408) 987-8080

      HELP                                      Program Device
   Change Disk                    Enter JEDEC file name [.JED]: DICE1.JED
 Edit JEDEC File                  Directory of .JED files for: C:\IPLS
 Program Device
  Verify Device
 Examine Device
      EXIT
                                          Press <-- to use default name
```

Screen 9.



```
JEDEC File: DICE1.JED                 Device: 5C060            LOGIC PROGRAMMER

        LOGIC P                     JEDEC File Header Text
        Copyrig
        Copyrig
        3065 Bo  Designer: Your Name
                 Company: Your Company
                 Part #:
                 Revision: 0.0
              H  EPLD: 5C060
                 Device code:
          Chang
                 Comment: PART A: DICE ROLLING
        Edit JE   LB Version 3.0, Baseline 17x, 9/26/85

         Progra

         Verify  Insert a 5C060 into the socket
                 Strike any key when ready
         Examin

              E
```

Screen 10.

Since you already have a proven design of a rolling dice from part A, we shall use the Logic Builder and edit that design. You may wish to save the original design at this stage. You can do this by using the <F10> key in the Main Menu. Press <F10> and issue the following command before re-entering the iPLS menu:

```
COPY DICE1.* DICE1A.*
```

The truth table is shown in Table 6.

Now you can use the iPLDS to design and program the device.

Go through the same steps to program the device as in Part A of the design example. Use the Logic builder, the Logic Optimizing Compiler, and the Logic Programmer respectively. The Logic Optimizing Compiler and the Logic Programmer steps are identical to the corresponding steps explained in part A of the design example. However, the Logic Builder will be used to edit the existing file, DICE1, to include the switch feature as follows:

Invoke the Logic Builder Menu from the iPLS main menu by pressing the <F3> key. Once you obtain the Logic Builder Configuration Menu, type in DICE1 as your input file name.

Use (Shift)(F3) to get the Inputs option and then add switch at pin #2 to it.

Inputs: `CLOCK, SWITCH@2 <Enter>`

Now press <F2> to exit to the Logic Builder Main Menu and answer the prompts as given in Table 7.

All that is left to do now is to edit the four equations, IN1A, IN1B, IN1C, IN1D to add the SWITCH option to it. Edit the four equations as follows:

Edit Function

When you press the "Edit" function key, <F6>, while in the main menu, the edit menu is displayed on the left side of the screen as shown in Screen 11. If you wish to edit an EQN Primitive displayed on the screen, press <F6>. Then the equation is moved to the prompt line where it can be edited.

Hence, the Boolean expressions for this case would consider the situations of when the switch was ON as well as OFF. The Boolean equations would contain the expression for the switch as follows.

```
DICE1A = ((1A*/1B*/1C*/1D)+(1A**1B*/1C*/1D)
         +(1A*1B*1C*/1D)
         +(/1A*/1B*/1C*/1D))*/SWITCH
         +((/1A*1B*/1C*/1D)+(/1A*1B*1C*/1D)
         +(/1A*1B*1C*1D)
         +(/1A*/1B*/1C*/1D))*SWITCH

DICE1B = ((/1A*1B*/1C*/1D)+(1A*1B*/1C*/1D)
         +(/1A*1B*1C*/1D)+(1A*1B*1C*/1D)
         +(/1A*1B*1C*1D))*/SWITCH
         +((1A*/1B*/1C*/1D)
         +(/1A*1B*/1C*/1D)+(1A*1B*/1C*/1D)
         +(/1A*1B*1C*/1D)
         +(1A*1B*1C*/1D))*SWITCH

DICE1C = ((/1A*1B*1C*/1D)
         +(1A*1B*1C*/1D)
         +(/1A*1B*1C*1D))*/SWITCH
         +((1A*1B*/1C*/1D)+(/1A*1B*1C*/1D)
         +(1A*1B*1C*/1D))*SWITCH

DICE1D = (/1A*1B*1C*1D)*/SWITCH
         +(1A*1B*1C*/1D)*SWITCH
```

The equation primitive must be displayed on the screen in order to edit that equation. In order to display the equation on the screen, use the "Find" command, <F5>, to find it.

The "Find" command prompts for a node name: then searches the design for that node and displays it. If the direction arrow points to the left, the primitive on the output side of the node is shown. If the direction arrow points to the right, the first primitive on the input side is shown.

After you have modified all four equations to include the SWITCH feature, return to the iPLDS main menu using the <F5> key and save the design using the <F6> key. You can verify your ADF file with the ADF file for part B given in Appendix D.

The file is ready to be compiled using the LOC, and the device is ready to be programmed using the LP.

The steps required to use the LOC and the LP are identical to the steps in part A.

Now the device that has been programmed is ready to be tested. At this stage in the design, you have completed part B of the design which is to add a switch to give the roll/no-roll option.

The programmed device can be tested as described in Appendix C.

Let us summarize before moving on to the next part of the design.

Table 6. Truth Table for DICE1

	Input State				Output State							
SWITCH	1A	1B	1C	1D	1A	1B	1C	1D	1A	1B	1C	1D
					Valid state				Invalid state			
					REMAIN IN THE SAME STATE							
0	1	0	0	0	1	0	0	0(1)				
0	0	1	0	0	0	1	0	0(2)				
0	1	1	0	0	1	1	0	0(3)				
0	0	1	1	0	0	1	1	0(4)				
0	1	1	1	0	1	1	1	0(5)				
0	0	1	1	1	0	1	1	1(6)				
					CONTROL THE INVALID STATES							
0	0	0	1	0					0	0	0	0
0	1	0	1	0					0	0	0	0
0	0	0	0	1					0	0	0	0
0	1	0	0	1					0	0	0	0
0	0	1	0	1					0	0	0	0
0	1	1	0	1					0	0	0	0
0	0	0	1	1					0	0	0	0
0	1	1	1	1					0	0	0	0
0	0	0	0	0					1	0	0	0(1)
					CHANGE TO THE NEXT VALID STATE*							
1	1	0	0	0(1)	0	1	0	0(2)				
1	0	1	0	0(2)	1	1	0	0(3)				
1	1	1	0	0(3)	0	1	1	0(4)				
1	0	1	1	0(4)	1	1	1	0(5)				
1	1	1	1	0(5)	0	1	1	1(6)				
1	0	1	1	1(6)	1	0	0	0(1)				
					CONTROL THE INVALID STATES							
1	0	0	1	0				0	0	0	0	
1	1	0	1	0				0	0	0	0	
1	0	0	0	1				0	0	0	0	
1	1	0	0	1				0	0	0	0	
1	0	1	0	1				0	0	0	0	
1	1	1	0	1				0	0	0	0	
1	0	0	1	1				0	0	0	0	
1	1	0	1	1				0	0	0	0	
1	1	1	1	1				0	0	0	0	
1	0	0	0	0				1	0	0	0(1)	

Note: This part of the truth table is identical to Table 3.

We have briefly discussed the EPLD and the IPLDS family of parts. We have also defined the design problem. We have implemented the design using the state equations and the truth table, edited an existing design to add features, and actually programmed a device using the Logic Builder, Logic Optimizing Compiler, and the Logic Programmer.

Our logic in implementing the dice example is to use the LED pairs in outputs 1A, 1B, 1C, and 1D respectively as

Table 7.

Prompts	User Entry
`Select a primitive for switch@2 to drive:`	`INP <Enter>`
`Out:`	`SWITCH<Enter>`
`Select a primitive for switch to drive:`	`EQN<Enter>`

shown in Figure 8. These LEDs are lit up to generate numbers between 1 and 6 inclusive. We are using a D-TYPE FLIP-FLOP to implement the truth table. The clock is a free running clock. A push button switch is also supplied to give the roll/no-roll option. Whenever the switch is ON, the LEDs roll, and when the switch is OFF, the LEDs display a number between 1 and 6, as long as the clock is supplied to the device.

After seeing the dice roll and display a number, you can either quit or move onto parts C, D and E of the design process. The following three parts describe a versatile use of the EPLD concept.

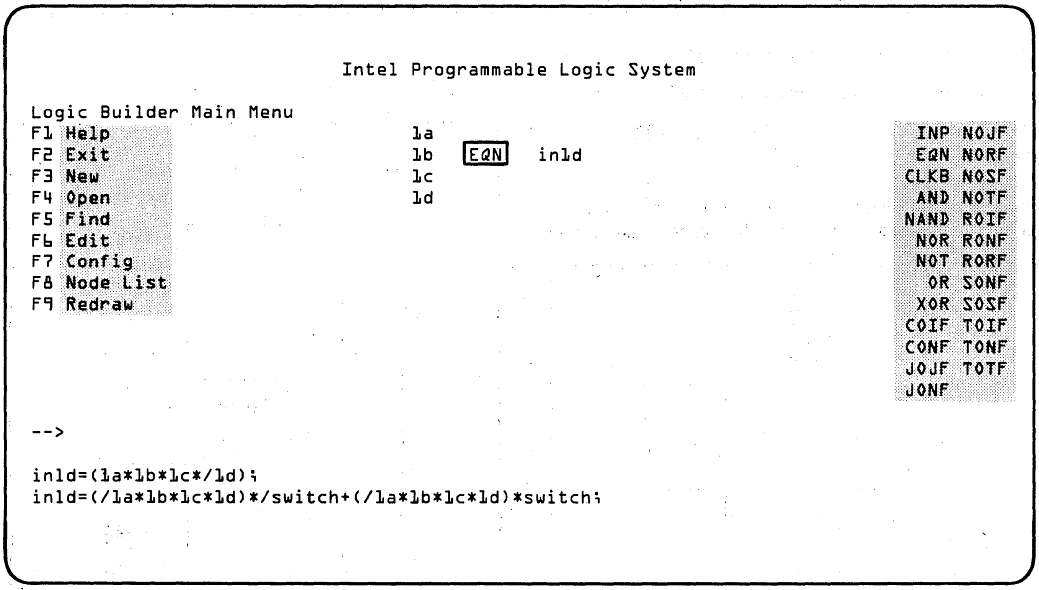
PART C

We are using an EPLD 5C060 which is a 24 pin, 600 gate device. It has four dedicated input pins and 16 input/output pins. Up to this point you have used only one input pin which is the switch and only four input/output pins for the four LEDs 1A, 1B, 1C, 1D.

Part C of the design is to include a second dice with the first dice. This is a step towards real-world application since dice are usually rolled in pairs. At the end of this section, you will have a pair of dice rolling and displaying a pair of numbers. All the conditions and truth tables and Boolean expressions that were designed for part B, hold good for DICE1. The equations for DICE2 would change slightly as explained below.

You have designed a 6 state counter and can define a carry out (fortunately you can use state 6 and do not require extra logic). You can use the carry out as an enable input to form two cascaded counters.

The carry out of 1D is used as an enable input to DICE2. Hence, 1D performs the same function as the push button switch performed in dice 1. Therefore, whenever 1D is enabled or logic high, DICE2 is enabled and rolls a number. DICE2 displays the number when 1D is disabled or logic is low. This configuration is shown in Figure 9.


```
                    Intel Programmable Logic System

Logic Builder Main Menu
F1 Help                       1a                                   INP  NOJF
F2 Exit                       1b  EQN   in1d                       EQN  NORF
F3 New                        1c                                  CLKB  NOSF
F4 Open                       1d                                   AND  NOTF
F5 Find                                                           NAND  ROIF
F6 Edit                                                            NOR  RONF
F7 Config                                                          NOT  RORF
F8 Node List                                                        OR  SONF
F9 Redraw                                                          XOR  SOSF
                                                                  COIF  TOIF
                                                                  CONF  TONF
                                                                  JOJF  TOTF
                                                                  JONF

-->

in1d=(1a*1b*1c*/1d);
in1d=(/1a*1b*1c*1d)*/switch+(/1a*1b*1c*1d)*switch;
```

Screen 11.

Table 8.

PROMPTS	USER ENTRY
Find: (Now use the <cursor left> key to obtain the EQN Primitive.) Edit: IN1D = (/1A*1B*1C*1D)*/SWITCH+(1A*1B*1C*/1D)*SWITCH;<Enter>	IN1D <Enter>
Find: (Now use the <cursor left> key to obtain the EQN Primitive.) Edit: IN1C = ((/1A*1B*1C*/1D)+(1A*1B*1C*/1D)+(/1A*1B*1C*1D))*/SWITCH +((1A*1B*/1C*/1D)+(/1A*1B*1C*/1D)+(1A*1B*1C*/1D))*SWITCH; <Enter>	IN1C <Enter>
Find: (Now use the <cursor left> key to obtain the EQN Primitive.) Edit: IN1B = ((/1A*1B*/1C*/1D)+(1A*1B*/1C*/1D)+(/1A*1B*1C*1D)+(1A*1B*1C*/1D) +(/1A*1B*1C*1D))*/SWITCH +((1A*/1B*/1C*/1D)+(/1A*1B*/1C*/1D)+(1A*1B*/1C*/1D)+ (/1A*1B*1C*/1D)+(1A*1B*1C*/1D))*SWITCH;<Enter>	IN1B <Enter>
Find: (Now use the <cursor left> key to obtain the EQN Primitive.) Edit: IN1A=((1A*/1B*/1C*/1D)+(1A*1B*/1C*/1D)+(1A*1B*1C*/1D)+ (/1A*/1B*/1C*/1D))*/SWITCH+((/1A*1B*/1C*/1D)+(/1A*1B*1C*/1D) +(/1A*1B*1C*1D)+(/1A*/1B*/1C*/1D))*SWITCH;<Enter>	IN1A <Enter>

The two conditions obtained are as follows:

When power is ON and 1D is enabled, DICE2 will roll.

When power is ON and 1D is disabled, DICE2 will display.

For DICE1, the logic conditions remain the same as in part A. Just as you used the switch to enable and disable

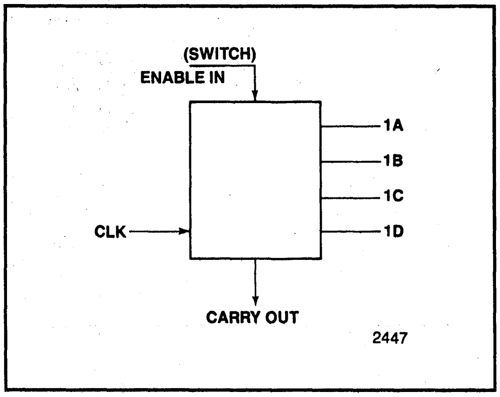


Figure 8.

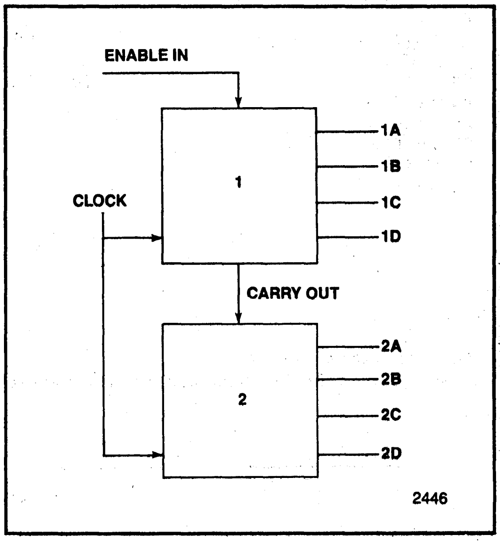


Figure 9.

DICE1, you will use the switch as well as the output of LED 1D to enable and disable DICE2; because the number on DICE2 is a function of both the switch and the present state of LED 1D, as explained above.

Now write down the truth table since the state diagrams can easily be inferred from the truth table. Please note that the truth table is identical to the one for DICE1 except for the switch input. For DICE2, you will have the combination of the switch and the 1D, as shown in Table 9.

The Boolean expressions for part C will consider the situation when the switch is ON as well as OFF and also 1D enabled or disabled respectively. The Boolean equations will contain the expression for the switch and LED 1D, as shown below.

```
DICE2A = ((2A*/2B*/2C*/2D)+(2A*2B*/2C*/2D)
          +(2A*2B*2C*/2D)+(/2A*/2B*/2C*/2D))
          *(/SWITCH*/1D)
          +((/2A*2B*/2C*/2D)+(/2A*2B*2C*/2D)
          +(/2A*2B*2C*2D)+(/2A*/2B*/2C*/2D))
          *(SWITCH*1D)

DICE2B = ((/2A*2B*/2C*/2D)+(2A*2B*/2C*/2D)
          +(/2A*2B*2C*/2D)+(2A*2B*2C*/2D)
          +(/2A*2B*2C*2D))*(/SWITCH*/1D)
          +((2A*/2B*/2C*/2D)
          +(/2A*2B*/2C*/2D)+(2A*2B*/2C*/2D)
          +(/2A*2B*2C*/2D)
          +(2A*2B*2C*/2D))*(SWITCH*1D)

DICE2C = ((/2A*2B*2C*/2D)+(2A*2B*2C*/2D)
          +(/2A*2B*2C*2D))*(/SWITCH*/1D)
          +((2A*2B*/2C*/2D)
          +(/2A*2B*2C*/2D)+(2A*2B*2C*/2D))
          *(SWITCH*1D)

DICE2D = (/2A*2B*2C*2D)*(/SWITCH*/1D)
          +(2A*2B*2C*/2D)*(SWITCH*1D)
```

Now you can use the iPLDS to program and test the device as explained in appendix C. At this stage in design, you have completed part C of the design which is to add a second DICE to the first one giving the the roll/no-roll option.

In part C of the design process, you have used one dedicated input which is the switch, and a total of eight output pins for the two pairs of LEDs, 1A, 1B, 1C, 1D and 2A, 2B, 2C, 2D respectively. You have also used the RORF primitive, since the design logic was the same for DICE2 as it was for DICE1. This leaves 3 dedicated inputs and 8 I/O pins on the 5C060 device.

You can stop the design now or go onto part D which gives the next option, which is adding the spin.

PART D

This is the fourth step in our design process and adds the spin option to the two dice that are rolling when the switch is pushed and display a number when the switch is released. The logic used to implement the spin concept is as follows:

When the power is ON and the switch is OFF, DICE1 and DICE2 display a random number according to the logic defined in parts B and C respectively.

But, when power is ON and the switch is ON, the two dice spin by lighting the LEDs B, C, and D. That is, DICE1 will light LEDs 1B, 1C, 1D while DICE2 will light LEDs 2B, 2C, and 2D. This pattern on the LEDs will generate the spinning pattern. The logic is shown in the truth table in Table 10. The schematic is shown in Figure 10.

As you can see from the truth table, when the present state is any of the three valid states, then the two dice will spin. The dice will also spin if the present state is an invalid state, because all the invalid states go to "0 0 0 0" in the next state. But from the truth table in Table 10, you see that this particular state is a valid state lighting LED C.

The spin frequency should be chosen to be visually appealing and should be high enough to ensure randomness of the dice. If we use the "carry out" state of DICE2, then the spin pattern will only change once for every combination of the two dice. This will ensure randomness. The "carry out" of DICE2 is signal 2d; we do not need extra terms to derive it.

Thus we have achieved our objective of adding the spinning option to the two dice.

The Boolean equations that are obtained from the above truth table are as follows:

```
SPIN1B = (SWITCH*2d*/S1D*/S1C*/S1B*S1A)

SPIN1C = (SWITCH*2d*/S1D*/S1C*/S1B*/S1A)

SPIN1D = (SWITCH*2d*/S1D*S1C*/S1B*/S1A)

SPIN2B = (SWITCH*2d*/S2D*/S2C*/S2B*S2A)

SPIN2C = (SWITCH*2d*/S2D*/S2C*/S2B*/S2A)

SPIN2D = (SWITCH*2d*/S2D*S2C*/S2B*/S2A)
```

Please note in the above equations that A, B, C, and D refer to both DICE1 and DICE2. For DICE1 the above set of equations would be 1A, 1B, 1C, and 1D. For DICE2 the above set of equations would be 2A, 2B, 2C, and 2D respectively. SD is the feedback obtained from IN D of both DICE1 and DICE2 respectively. If the switch is not ON, the dice will not spin and a random pair of numbers will be displayed by the two dice; but, if the switch is ON, then the two dice will spin according to the truth table and Boolean expression given in Table 10.

Table 9. Truth Table for DICE2

Input State					Output State							
(SWITCH*1D)	2A	2B	1C	2D	2A	2B	2C	2D	2A	2B	2C	2D
					Valid state				Invalid state			
REMAIN IN THE SAME STATE												
0	1	0	0	0	1	0	0	0(1)				
0	0	1	0	0	0	1	0	0(2)				
0	1	1	0	0	1	1	0	0(3)				
0	0	1	1	0	0	1	1	0(4)				
0	1	1	1	0	1	1	1	0(5)				
0	0	1	1	1	0	1	1	1(6)				
CONTROL THE INVALID STATES												
0	0	0	1	0					0	0	0	0
0	1	0	1	0					0	0	0	0
0	0	0	0	1					0	0	0	0
0	1	0	0	1					0	0	0	0
0	0	1	0	1					0	0	0	0
0	1	1	0	1					0	0	0	0
0	0	0	1	1					0	0	0	0
0	1	1	1	1					0	0	0	0
0	0	0	0	0					1	0	0	0(1)
CHANGE TO THE NEXT VALID STATE*												
1	1	0	0	0(1)	0	1	0	0(2)				
1	0	1	0	0(2)	1	1	0	0(3)				
1	1	1	0	0(3)	0	1	1	0(4)				
1	0	1	1	0(4)	1	1	1	0(5)				
1	1	1	1	0(5)	0	1	1	1(6)				
1	0	1	1	1(6)	1	0	0	0(1)				
CONTROL THE INVALID STATES												
1	0	0	1	0					0	0	0	0
1	1	0	1	0					0	0	0	0
1	0	0	0	1					0	0	0	0
1	1	0	0	1					0	0	0	0
1	0	1	0	1					0	0	0	0
1	1	1	0	1					0	0	0	0
1	0	0	1	1					0	0	0	0
1	1	0	1	1					0	0	0	0
1	1	1	1	1					0	0	0	0
1	0	0	0	0					1	0	0	0(1)

Note the extreme similarity between this truth table and the one given in Table 3.

Table 10. Truth Table to Spin Two Dice

Input State					Output State			
SWITCH	A	B	C	D	A	B	C	D
CHANGE TO THE NEXT VALID STATE								
1	0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0	0
1	0	0	1	0	0	0	0	1
1	0	1	0	0	0	0	0	0
ROLLING INTO A VALID STATE								
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	0
1	1	1	0	1	0	0	0	0
1	0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0	0
1	0	1	1	1	0	0	0	0
1	1	1	1	1	0	0	0	0

We have chosen the following two primitives for part D:

Registered Output Registered Feedback (RORF)

No output JK Feedback (NOJF)

For the dice spinning option you will use the RORF and for the dice not spinning option you will use the NOJF, while using the Logic Builder.

When you add the spinning option to the pair of rolling dice, you obtain the following boolean equations. (These Boolean equations satisfy the requirements of the two dice spinning when the switch is on and displaying a number when the switch is off).

```
SPIN1A = (/SWITCH*1A)
SPIN1B = (/SWITCH*1B)
         + (SWITCH*2d*/S1D*/S1C*/S1B*S1A)
SPIN1C = (/SWITCH*1C)
         + (SWITCH*2d*/S1D*/S1C*/S1B*/S1A)
SPIN1D = (/SWITCH*1D)
         +(SWITCH*2d*/S1D*S1C*/S1B*/S1A)
SPIN2A = (/SWITCH*2A)
SPIN2B = (/SWITCH*2B)
         + (SWITCH*2d*/S2D*/S2C*/S2B*S2A)
SPIN2C = (/SWITCH*2C)
         + (SWITCH*2d*/S2D*/S2C*/S2B*/S2A)
SPIN2D = (/SWITCH*2D)
         +(SWITCH*2d*/S2D*S2C*/S2B*/S2A)
```

At the end of the design step, you have completed all the design steps. You can now program the device using iPLDS.

The correct ADF file is included in Appendix D for your reference. You can refer to it to verify the ADF file you have created.

The programmed device can be tested on:

- A PCB with slow clock

For information on this board and on testing your design, please refer to Appendix C.

It works!

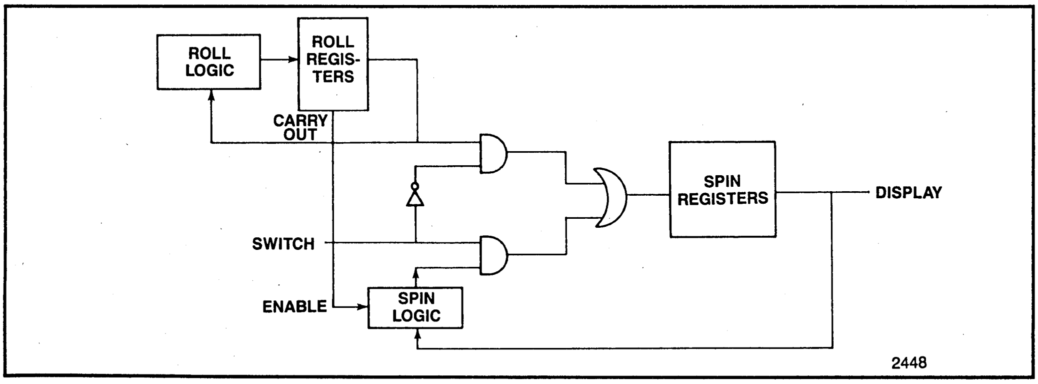


Figure 10.

LATE NEWS FLASH

The PCBs have been made and we have units in the field. Now Marketing wants the design updated! Field trials of the dice showed that the battery needed to last longer. A simple mod to the design, chop the drive to the LEDs, extends the battery life.

This is very simple using the EPLDs. Reprogram the EPLD and test it. Imagine how difficult it would have been without using EPLDs.

PART E

This step of the design process is to modify the existing circuit to add the power save feature which will extend the battery life. This can easily be done by chopping the drive to the LEDs. Chopping the drive to the LEDs can be done as follows:

When you designed the circuit and implemented it using the iPLS, you have set the output enable (Oe) to VCC supply. This means that the LEDs are enabled 100% of the time. You can "chop" the drive to the LEDs with a conveniant high (above 50Hz) signal that will not be visible to the human eye.

Next set Output enable (Oe) to the clock signal. Thus, depending on the clock input the LEDs will only be on 50% of the time and battery life is extended as required. You can easily modify the ADF file to change the Oe input from VCC to CLOCK and then test the design using the PCB as explained in Appendix C.

CONCLUSION

You should now have a comprehensive knowledge of Intel's EPLD and iPLDS family of devices.

With this knowledge you will be able to implement designs using the iPLDS tools.

Good Luck!

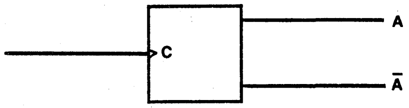
APPENDIX A:
BASIC DEFINITIONS

BASIC DEFINITIONS

Logic Design — A systematic procedure for realizing specified terminal characterisitics of digital networks, at either the device or system level.

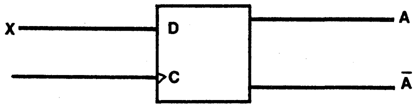
CLOCKED FLIP-FLOP — Output determined by the leading or trailing edge of clock pulse.

T FLIP-FLOP — Output changes value with every input clock pulse.



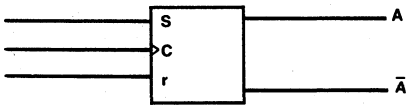
T FLIP-FLOP

D FLIP-FLOP — Output determined by the input signal when clock pulse present.



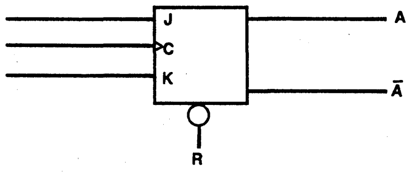
D FLIP-FLOP

S-R FLIP-FLOP — Output states synchronized with the clock pulse and controlled by the input signals, S and R.



S-R FLIP-FLOP

J-K FLIP-FLOP — Output states synchronized with the clock pulse and controlled by the input signals, J and K.



J-K FLIP-FLOP

COMBINATORIAL CIRCUIT — Output determined by current value of input signal.

REGISTERED CIRCUIT — Output determined by sequence of input signals.

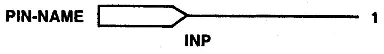
Intel Schematic Primitive — One of the basic functional blocks needed to design circuits for Intel programmable logic products.

Truth Table — A list of all the input-output possibilities of a logic circuit.

Boolean Logic — Describes logic that obeys the theorems of Boolean algebra. The Boolean portion of a design is that portion which can be implemented in the AND-OR matrix.

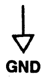
State Diagram — A diagram that shows the succession of output states through which the circuit passes as its input signals vary.

INP — Input



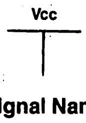
Input Primitive

GND — Ground.

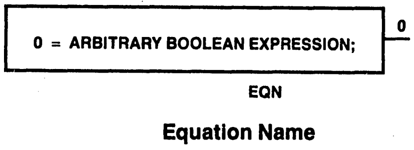


Ground Signal Name

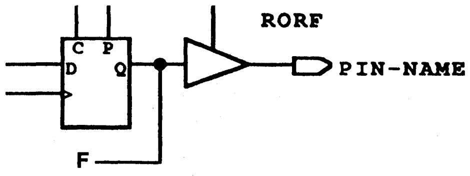
VCC — Signal



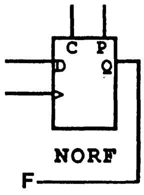
EQN — Equation



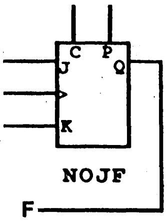
Registered Output Registered Feedback (RORF)



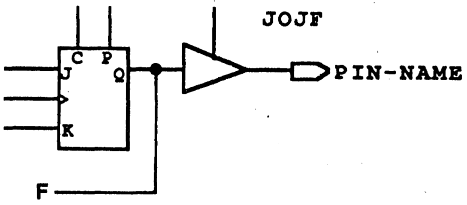
No Output Registered Feedback (NORF)



No Output JK Feedback (NOJF)



JK Output JK Feedback (JOJF)



Security Bit — A feature that prevents the device from being interrogated or being accidentally programmed.

Turbo-bit — A control bit that allows you to choose the speed and power characteristics of the device. If the inputs are static for approximately 50 ns and the Turbo-bit is not programmed, the device will enter power down mode. When the input changes, the device will take an extra 3-5 ns to wake-up and react to the change. Programming the Turbo-bit inhibits the power down.

Macrocell — A basic building block of Intel's programmable logic devices. A macrocell consists of two sections: combinatorial logic and output logic. The combinatorial logic allows a wide variety of logic functions. The output logic has two data paths: one leads to the other macrocells or feeds back to the macrocell itself: the other is configured as a pin configuration acting as input, output, or bi-directional I/O port on the chip.

Node — A wire connecting two or more primitives in a schematic.

Pin — A node that is connected to an input or I/O primitive on one end and a pin of the chip on the other end.

Product tem (P-Term) — Two or more factors in a boolean expression combined with the AND operator consitutes a logic product term.

JEDEC Standard File — An industry-wide standard for the transfer of information between a data preparation system and a logic device programmer.

EPLD PROGRAMMING TECHNIQUES

You can enter your design in the following ways

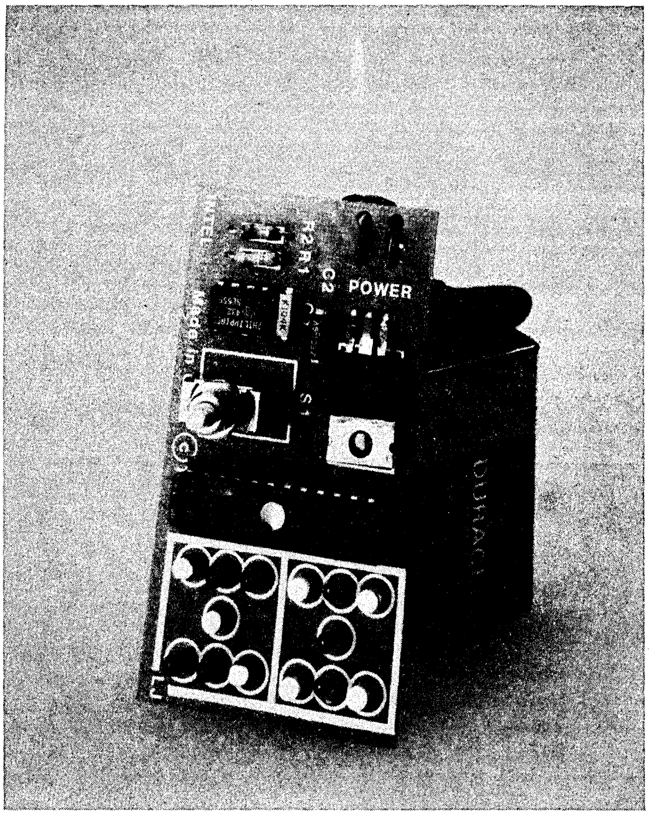
1. BOOLEAN EQUATION — entering the design in BOOLEAN equations or expressions.
2. NETLIST CAPTURE — selecting components and specifying interconnections until all elements are specified.
3. SCHEMATIC CAPTURE — using a mouse and menu driven environment.
4. STATE MACHINE — specifying states and conditional branches and also inputs/outputs to the state machines.

APPENDIX B: COMPONENTS LIST

COMPONENTS USED IN DESIGN

In order to implement the EPLD program, you should use the following:

- An 5C060 EPLD
- A pair of seven discrete LEDs (Dice 1, Dice 2)
- A timer to generate a clock signal (NE555)
- A voltage regulator to generate a fixed voltage of 5 volts (7805)
- A push button switch to control the spinning mechanism
- A 9-Volt DC battery source to generate the power supply
- Capacitors C1 = 0.1 MF, C2 = 0.01 MF
- Resistors R1 = 390K, R2 = 100K
- A PCB as explained in Appendix C



APPENDIX C: PCB DESCRIPTION

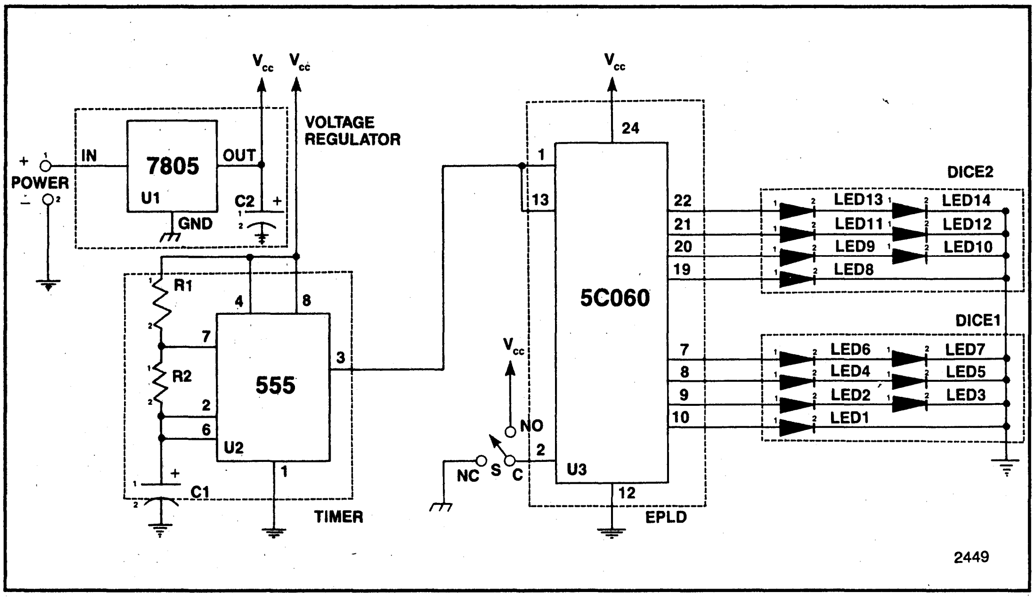


Figure C-1

You can test each part of your design using the PCB with a slow clock on it.

The PCB is a board that is very specific to the dice example. The PCB is portable, approximately 2″ × 3″. All the components except for the EPLD are easily available commercially. A complete list of all the components that are required for the PCB is given in Appendix B. The circuit can easily be connected and tested using the circuit diagram given below. After the four steps of the design are completed, the PCB can be used to throw a pair of dice in any home games such as Monopoly etc.

After the EPLD is programmed using the Logic Programmer, it can be inserted into the PCB. For design steps B, C, and D the push button switch can be used to generate the roll/no-roll or the spin/no spin option.

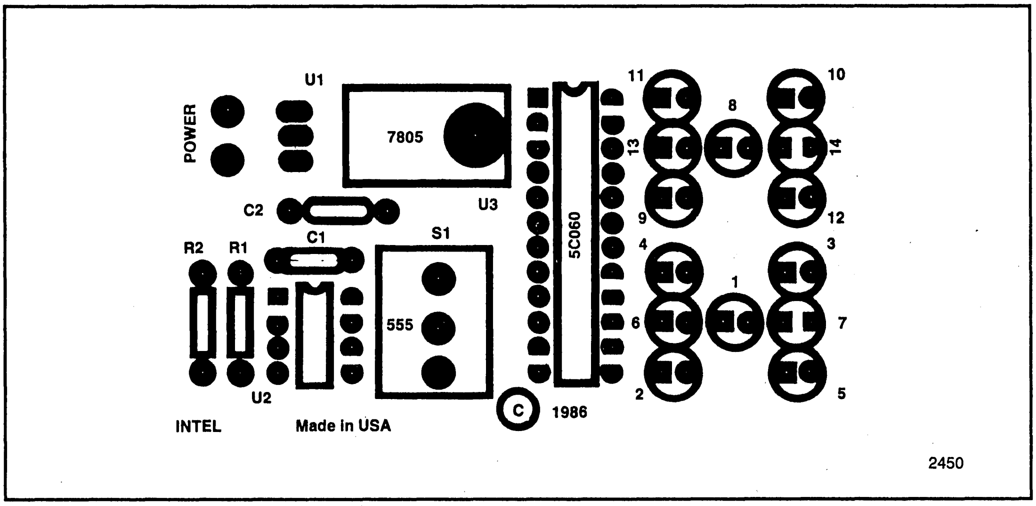

POWER
U1
7805
U3
C2
C1
R2
R1
S1
555
U2
5C060
INTEL
Made in USA
© 1986
11
10
8
13
14
9
12
4
3
1
6
7
2
5
2450

Figure C-2

APPENDIX D

```
ADF FOR PART A: SINGLE DICE ROLLING
-----------------------------------

Lakshmi Jayanthi
DSO Applications
February 19, 1986

5C060

Part A:   DICE ROLLING

LB Version 3.0, Baseline 17x, 9/26/85
PART: 5C060

INPUTS: clock1

OUTPUTS: dice1a@10,dice1b@9,dice1c@8,dice1d@7

NETWORK:

dice1a,1a = RORF (in1a,clock1,GND,GND,VCC)
dice1b,1b = RORF (in1b,clock1,GND,GND,VCC)
dice1c,1c = RORF (in1c,clock1,GND,GND,VCC)
dice1d,1d = RORF (in1d,clock1,GND,GND,VCC)

clock1 = INP (clock1)

EQUATIONS:

in1a =(/1a*1b*/1c*/1d)
         +(/1a*1b*1c*/1d)
         +(/1a*1b*1c*1d)
         +(/1a*/1b*/1c*/1d);
in1b =(1a*/1b*/1c*/1d)
         +(/1a*1b*/1c*/1d)
         +(1a*1b*/1c*/1d)
         +(/1a*1b*1c*/1d)
         +(1a*1b*1c*/1d);
in1c =(1a*1b*/1c*/1d)
         +(/1a*1b*1c*/1d)
         +(1a*1b*1c*/1d);
in1d =(1a*1b*1c*/1d);

END$
```

```
RPT FOR PART A: SINGLE DICE ROLLING
------------------------------------

Logic Optimizing Compiler Utilization Report

 ***** Design implemented successfully

Lakshmi Jayanthi
DSO Applications
February 19, 1986

5C060

Part A:   DICE ROLLING

LB Version 3.0, Baseline 17x, 9/26/85

                  5C060
              - - - - -
    clock1  -| 1  .  24|- Vcc
       GND  -| 2     23|- GND
       GND  -| 3     22|- GND
       GND  -| 4     21|- GND
       GND  -| 5     20|- GND
       GND  -| 6     19|- GND
    dice1d  -| 7     18|- GND
    dice1c  -| 8     17|- GND
    dice1b  -| 9     16|- GND
    dice1a  -|10     15|- GND
       GND  -|11     14|- GND
       GND  -|12     13|- GND
              - - - - -


**INPUTS**

                                                     |            Feeds:
       Name   Pin   Resource    MCell #      PTerms  |   MCells      OE   Clear   Clock

     clock1    1        INP          -            -           -       -       -    CLK1

**OUTPUTS**

                                                     |            Feeds:
       Name   Pin   Resource    MCell #      PTerms  |   MCells      OE   Clear   Clock

     dice1d    7       RORF         13         1/ 8             13      -       -       -
                                                                14
                                                                15
                                                                16

     dice1c    8       RORF         14         2/ 8             13      -       -       -
                                                                14
                                                                15
                                                                16

     dice1b    9       RORF         15         2/ 8             13      -       -       -
                                                                14
                                                                15
                                                                16

     dice1a   10       RORF         16         2/ 8             13      -       -       -
                                                                14
                                                                15
                                                                16
```

```
**UNUSED RESOURCES**

      Name   Pin   Resource      MCell     PTerms

       -      2       -            -          -
       -      3       -            9          8
       -      4       -           10          8
       -      5       -           11          8
       -      6       -           12          8
       -     11       -            -          -
       -     13       -            -          -
       -     14       -            -          -
       -     15       -            8          8
       -     16       -            7          8
       -     17       -            6          8
       -     18       -            5          8
       -     19       -            4          8
       -     20       -            3          8
       -     21       -            2          8
       -     22       -            1          8
       -     23       -            -          -

**PART UTILIZATION**

22%     Pins
25%     MacroCells
5%      Pterms
```

NOTE: Since part A is a simple design, the part utilization is very low.

```
ADF FOR PART B: SINGLE DICE ROLL/NOT ROLL
------------------------------------------------------

Lakshmi Jayanthi
DSO Applications
February 19, 1986


5C060

PART B: DICE ROLL AND NOT ROLL

LB Version 3.0, Baseline 17x, 9/26/85
PART: 5C060

INPUTS: clock1,switch@2

OUTPUTS: dice1a@10,dice1b@9,dice1c@8,dice1d@7

NETWORK:

dice1a,1a = RORF (in1a,clock1,GND,GND,VCC)
dice1b,1b = RORF (in1b,clock1,GND,GND,VCC)
dice1c,1c = RORF (in1c,clock1,GND,GND,VCC)
dice1d,1d = RORF (in1d,clock1,GND,GND,VCC)

clock1 = INP (clock1)

switch = INP(switch)

EQUATIONS:

in1a =(/1a*/1b*/1c*/1d*/switch)
        +(1a*/1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(/1a*1b*1c*1d*switch);
in1b =(/1a*1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1c =(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1d =(/1a*1b*1c*1d*/switch)
        +(1a*1b*1c*/1d*switch);

END$
```

```
RPT FOR PART B: SINGLE DICE ROLL/NOT ROLL
-------------------------------------------


Logic Optimizing Compiler Utilization Report

 ***** Design implemented successfully

Lakshmi Jayanthi
DSO Applications
February 19, 1986


5C060

PART B: DICE ROLL AND NOT ROLL

LB Version 3.0, Baseline 17x, 9/26/85

             5C060
          - - - - -
clock1 -| 1    24|- Vcc
switch -| 2    23|- GND
   GND -| 3    22|- GND
   GND -| 4    21|- GND
   GND -| 5    20|- GND
   GND -| 6    19|- GND
dice1d -| 7    18|- GND
dice1c -| 8    17|- GND
dice1b -| 9    16|- GND
dice1a -|10    15|- GND
   GND -|11    14|- GND
   GND -|12    13|- GND
          - - - - -

**INPUTS**

                                                   |        Feeds:
      Name   Pin   Resource    MCell #    PTerms   |   MCells       OE   Clear   Clock

    clock1    1        INP          -          -          -        -       -    CLK1

    switch    2        INP          -          -         13        -       -       -
                                                         14
                                                         15
                                                         16


**OUTPUTS**

                                                   |        Feeds:
      Name   Pin   Resource    MCell #    PTerms   |   MCells       OE   Clear   Clock

    dice1d    7       RORF         13       2/ 8         13        -       -       -
                                                         14
                                                         15
                                                         16

    dice1c    8       RORF         14       3/ 8         13        -       -       -
                                                         14
                                                         15
                                                         16
    dice1b     9        RORF          15        3/ 8            13      -       -
                                                                14
                                                                15
                                                                16

    dice1a    10        RORF          16        5/ 8            13      -       -
                                                                14
                                                                15
                                                                16


**UNUSED RESOURCES**

          Name   Pin   Resource        MCell      PTerms

           -      3        -             9           8
           -      4        -            10           8
           -      5        -            11           8
           -      6        -            12           8
           -     11        -             -           -
           -     13        -             -           -
           -     14        -             -           -
           -     15        -             8           8
           -     16        -             7           8
           -     17        -             6           8
           -     18        -             5           8
           -     19        -             4           8
           -     20        -             3           8
           -     21        -             2           8
           -     22        -             1           8
           -     23        -             -           -


**PART UTILIZATION**

27%       Pins
25%       MacroCells
10%       Pterms
```

NOTE: Part B of the design gets more complicated, hence the part utilization of the pins, macrocells and the Pterms is higher.

```
ADF FOR PART C: TWO DICE ROLLING
---------------------------------

Lakshmi Jayanthi
DSO Applications
February 19, 1986

5C060

PART C:  TWO DICE ROLL AND NOT ROLL

B Version 3.0, Baseline 17x, 9/26/85
PART: 5C060

INPUTS: clock1,clock2,switch@2

OUTPUTS: dice1a@10,dice1b@9,dice1c@8,dice1d@7,dice2a@19,dice2b@20,dice2c@21,dice
2d@22

NETWORK:

dice1a,1a = RORF (in1a,clock1,GND,GND,VCC)
dice1b,1b = RORF (in1b,clock1,GND,GND,VCC)
dice1c,1c = RORF (in1c,clock1,GND,GND,VCC)
dice1d,1d = RORF (in1d,clock1,GND,GND,VCC)

dice2a,2a = RORF (in2a,clock2,GND,GND,VCC)
dice2b,2b = RORF (in2b,clock2,GND,GND,VCC)
dice2c,2c = RORF (in2c,clock2,GND,GND,VCC)
dice2d,2d = RORF (in2d,clock2,GND,GND,VCC)

clock1 = INP (clock1)
clock2 = INP (clock2)

switch = INP (switch)

EQUATIONS:

in1a =(/1a*/1b*/1c*/1d*/switch)
        +(1a*/1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(/1a*1b*1c*1d*switch);


in1b =(/1a*1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1c =(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1d =(/1a*1b*1c*1d*/switch)
        +(1a*1b*1c*/1d*switch);
```

```
in2a =(/2a*/2b*/2c*/2d*/(1d*switch))
        +(2a*/2b*/2c*/2d*/(1d*switch))
        +(2a*2b*/2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*/2b*/2c*/2d*(1d*switch))
        +(/2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(/2a*2b*2c*2d*(1d*switch));
in2b =(/2a*2b*/2c*/2d*/(1d*switch))
        +(2a*2b*/2c*/2d*/(1d*switch))
        +(/2a*2b*2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*2b*2c*2d*/(1d*switch))
        +(2a*/2b*/2c*/2d*(1d*switch))
        +(/2a*2b*/2c*/2d*(1d*switch))
        +(2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));
in2c =(/2a*2b*2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*2b*2c*2d*/(1d*switch))
        +(2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));
in2d =(/2a*2b*2c*2d*/(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));

END$
```

```
RPT FOR PART C: TWO DICE ROLLING
--------------------------------

Logic Optimizing Compiler Utilization Report

 ***** Design implemented successfully

Lakshmi Jayanthi
DSO Applications
February 19, 1986

5C060

PART C:  TWO DICE ROLL AND NOT ROLL

B Version 3.0, Baseline 17x, 9/26/85

                  5C060
              - - - - -
   clock1  -| 1    24|- Vcc
   switch  -| 2    23|- GND
      GND  -| 3    22|- dice2d
      GND  -| 4    21|- dice2c
      GND  -| 5    20|- dice2b
      GND  -| 6    19|- dice2a
   dice1d  -| 7    18|- GND
   dice1c  -| 8    17|- GND
   dice1b  -| 9    16|- GND
   dice1a  -|10    15|- GND
      GND  -|11    14|- GND
      GND  -|12    13|- clock2
              - - - - -

**INPUTS**

                                                 |              Feeds:
   Name   Pin   Resource    MCell #     PTerms   |    MCells        OE   Clear   Clock

  clock1    1        INP          -          -             -         -       -    CLK1

  switch    2        INP          -          -             1         -       -       -
                                                           2
                                                           3
                                                           4
                                                          13
                                                          14
                                                          15
                                                          16

  clock2   13        INP          -          -             -         -       -    CLK2


**OUTPUTS**

                                                 |              Feeds:
   Name   Pin   Resource    MCell #     PTerms   |    MCells        OE   Clear   Clock

  dice1d    7       RORF         13       2/ 8             1         -       -       -
                                                           2
                                                           3
                                                           4
                                                          13
                                                          14
                                                          15
                                                          16
```

```
    dice1c    8       RORF         14        3/ 8          13      -       -       -
                                                           14
                                                           15
                                                           16

    dice1b    9       RORF         15        3/ 8          13      -       -       -
                                                           14
                                                           15
                                                           16

    dice1a    10      RORF         16        5/ 8          13      -       -       -
                                                           14
                                                           15
                                                           16

    dice2a    19      RORF          4        7/ 8           1      -       -       -
                                                            2
                                                            3
                                                            4

    dice2b    20      RORF          3        4/ 8           1      -       -       -
                                                            2
                                                            3
                                                            4

    dice2c    21      RORF          2        4/ 8           1      -       -       -
                                                            2
                                                            3
                                                            4

    dice2d    22      RORF          1        3/ 8           1      -
                                                            2
                                                            3
                                                            4


**UNUSED RESOURCES**

        Name   Pin   Resource      MCell     PTerms

         -       3       -            9         8
         -       4       -           10         8
         -       5       -           11         8
         -       6       -           12         8
         -      11       -            -         -
         -      14       -            -         -
         -      15       -            8         8
         -      16       -            7         8
         -      17       -            6         8
         -      18       -            5         8
         -      23       -            -         -

**PART UTILIZATION**

50%      Pins
50%      MacroCells
24%      Pterms
```

NOTE: in part C of the design you have added the second dice. Hence you can see that fifty percent of the device has been used.

```
ADF FOR PART D: TWO DICE SPINNING
-----------------------------------

Lakshmi Jayanthi
DSO Applications
February 19, 1986


5C060

PART D: TWO DICE SPINNING

B Version 3.0, Baseline 17x, 9/26/85
PART: 5C060

INPUTS: clock1,clock2,switch@2

OUTPUTS: spin1a@10,spin1b@9,spin1c@8,spin1d@7,spin2a@19,spin2b@20,spin2c@21,spir
2d@22

NETWORK:

1a = NOJF (in1a,clock1,in11a,GND,GND)
1b = NOJF (in1b,clock1,in11b,GND,GND)
1c = NOJF (in1c,clock1,in11c,GND,GND)
1d = NOJF (in1d,clock1,in11d,GND,GND)

2a = NOJF (in2a,clock2,in22a,GND,GND)
2b = NOJF (in2b,clock2,in22b,GND,GND)
2c = NOJF (in2c,clock2,in22c,GND,GND)
2d = NOJF (in2d,clock2,in22d,GND,GND)

in11a = NOT(in1a)
in11b = NOT(in1b)
in11c = NOT(in1c)
in11d = NOT(in1d)

in22a = NOT(in2a)
in22b = NOT(in2b)
in22c = NOT(in2c)
in22d = NOT(in2d)

spin1a,s1a = RORF (ins1a,clock1,GND,GND,VCC)
spin1b,s1b = RORF (ins1b,clock1,GND,GND,VCC)
spin1c,s1c = RORF (ins1c,clock1,GND,GND,VCC)
spin1d,s1d = RORF (ins1d,clock1,GND,GND,VCC)

spin2a,s2a = RORF (ins2a,clock2,GND,GND,VCC)
spin2b,s2b = RORF (ins2b,clock2,GND,GND,VCC)
spin2c,s2c = RORF (ins2c,clock2,GND,GND,VCC)
spin2d,s2d = RORF (ins2d,clock2,GND,GND,VCC)

clock1 = INP (clock1)
clock2 = INP (clock2)

switch = INP (switch)

EQUATIONS:

in1a =(/1a*/1b*/1c*/1d*/switch)
        +(1a*/1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(/1a*1b*1c*1d*switch);
in1b  =(/1a*1b*/1c*/1d*/switch)
        +(1a*1b*/1c*/1d*/switch)
        +(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*/1b*/1c*/1d*switch)
        +(/1a*1b*/1c*/1d*switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1c  =(/1a*1b*1c*/1d*/switch)
        +(1a*1b*1c*/1d*/switch)
        +(/1a*1b*1c*1d*/switch)
        +(1a*1b*/1c*/1d*switch)
        +(/1a*1b*1c*/1d*switch)
        +(1a*1b*1c*/1d*switch);
in1d  =(/1a*1b*1c*1d*/switch)
        +(1a*1b*1c*/1d*switch);

in2a  =(/2a*/2b*/2c*/2d*/(1d*switch))
        +(2a*/2b*/2c*/2d*/(1d*switch))
        +(2a*2b*/2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*/2b*/2c*/2d*(1d*switch))
        +(/2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(/2a*2b*2c*2d*(1d*switch));
in2b  =(/2a*2b*/2c*/2d*/(1d*switch))
        +(2a*2b*/2c*/2d*/(1d*switch))
        +(/2a*2b*2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*2b*2c*2d*/(1d*switch))
        +(2a*/2b*/2c*/2d*(1d*switch))
        +(/2a*2b*/2c*/2d*(1d*switch))
        +(2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));
in2c  =(/2a*2b*2c*/2d*/(1d*switch))
        +(2a*2b*2c*/2d*/(1d*switch))
        +(/2a*2b*2c*2d*/(1d*switch))
        +(2a*2b*/2c*/2d*(1d*switch))
        +(/2a*2b*2c*/2d*(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));
in2d  =(/2a*2b*2c*2d*/(1d*switch))
        +(2a*2b*2c*/2d*(1d*switch));

ins1a  = (/switch*1a);
ins1b  = (/switch*1b)
        +((2d*switch)*s1d*/s1c*/s1b*/s1a);
ins1c  = (/switch*1c)
        +((2d*switch)*/s1a*/s1b*/s1c*/s1d);
ins1d  = (/switch*1d)
        +((2d*switch)*/s1a*/s1b*s1c*/s1d);

ins2a  = (/switch*2a);
ins2b  = (/switch*2b)
        +((2d*switch)*s2d*/s2c*/s2b*/s2a);
ins2c  = (/switch*2c)
        +((2d*switch)*/s2a*/s2b*/s2c*/s2d);
ins2d  = (/switch*2d)
        +((2d*switch)*/s2a*/s2b*s2c*/s2d);

END$
```

```
LEF FOR PART D: TWO DICE SPINNING
---------------------------------

Lakshmi Jayanthi
DSO Applications
February 19, 1986


5C060

PART:
          5C060

INPUTS:

          clock1, clock2, switch@2


OUTPUTS:

          spin1a@10, spin1b@9, spin1c@8, spin1d@7, spin2a@19, spin2b@20,
          spin2c@21, spin2d@22

NETWORK:

          clock1 = INP(clock1)
          clock2 = INP(clock2)

          switch = INP(switch)

          spin1a, s1a = RORF(ins1a, clock1, GND, GND, VCC)
          spin1b, s1b = RORF(ins1b, clock1, GND, GND, VCC)
          spin1c, s1c = RORF(ins1c, clock1, GND, GND, VCC)
          spin1d, s1d = RORF(ins1d, clock1, GND, GND, VCC)

          spin2a, s2a = RORF(ins2a, clock2, GND, GND, VCC)
          spin2b, s2b = RORF(ins2b, clock2, GND, GND, VCC)
          spin2c, s2c = RORF(ins2c, clock2, GND, GND, VCC)
          spin2d, s2d = RORF(ins2d, clock2, GND, GND, VCC)

          %  ***  Resource, NOJF, was minimized to NORF  ***  %

          2d = NORF(..SG007D, clock2, GND, GND)

          %  ***  Resource, NOJF, was minimized to NOTF  ***  %

          2c = NOTF(..SG006D, clock2, GND, GND)

          %  ***  Resource, NOJF, was minimized to NORF  ***  %

          2b = NORF(..SG005D, clock2, GND, GND)

          %  ***  Resource, NOJF, was minimized to NORF  ***  %

          2a = NORF(..SG004D, clock2, GND, GND)

          %  ***  Resource, NOJF, was minimized to NORF  ***  %

          1d = NORF(..SG003D, clock1, GND, GND)

          %  ***  Resource, NOJF, was minimized to NORF  ***  %

          1c = NORF(..SG002D, clock1, GND, GND)
```

```
        %  ***  Resource, NOJF, was minimized to NORF  ***  %

        1b = NORF(..SG001D, clock1, GND, GND)

        %  ***  Resource, NOJF, was minimized to NORF  ***  %

        1a = NORF(..SG000D, clock1, GND, GND)

EQUATIONS:

        ins2d = switch' * 2d
              + 2d * switch * s2a' * s2b' * s2c * s2d';

        ins2c = switch' * 2c
              + 2d * switch * s2a' * s2b' * s2c' * s2d';

        ins2b = switch' * 2b
              + 2d * switch * s2d * s2c' * s2b' * s2a';

        ins2a = switch' * 2a

        ins1d = switch' * 1d
              + 2d * switch * s1a' * s1b' * s1c * s1d';

        ins1c = switch' * 1c
              + 2d * switch * s1a' * s1b' * s1c' * s1d';

        ins1b = switch' * 1b
              + 2d * switch * s1d * s1c' * s1b' * s1a';

        ins1a = switch' * 1a;

        ..SG000D = 1a' * 1b' * 1c' * 1d'
                 + 1a * 1c' * 1d' * switch'
                 + 1a' * 1b * 1d' * switch
                 + 1a * 1b * 1d' * switch'
                 + 1a' * 1b * 1c * switch;

        ..SG001D = 1b * 1d'
                 + 1b * 1a' * 1c * switch'
                 + 1a * 1c' * 1d' * switch;

        ..SG002D = 1c * 1b * 1d'
                 + 1c * 1a' * 1b * switch'
                 + 1a * 1b * 1d' * switch;

        ..SG003D = 1d * 1a' * 1b * 1c * switch'
                 + 1d' * 1a * 1b * 1c * switch;

        ..SG004D = 2a' * 2b' * 2c' * 2d'
                 + 2a * 2c' * 2d' * 1d'
                 + 2a * 2c' * 2d' * switch'
                 + 2a * 2b * 2d' * 1d'
                 + 2a * 2b * 2d' * switch'
                 + 2a' * 2b * 2d' * 1d * switch
                 + 2a' * 2b * 2c * 1d * switch;

        ..SG005D = 2b * 2d'
                 + 2b * 2a' * 2c * 1d'
                 + 2b * 2a' * 2c * switch'
                 + 2a * 2c' * 2d' * 1d * switch;
        ..SG006D = 2c * 2b'
                 + 2c * 2a * 2d
                 + 2c * 2d * 1d * switch
                 + 2c' * 2a * 2b * 2d' * 1d * switch;

        ..SG007D = 2d * 2a' * 2b * 2c * switch'
                 + 2d * 2a' * 2b * 2c * 1d'
                 + 2d' * 2a * 2b * 2c * 1d * switch;

END$
```

NOTE: PLease note how the IPLS software has simplified the equations for you. You need not worry about minimization. The complicated Boolean expressions have been minimized to a great extent.

```
RPT FOR PART D: TWO DICE SPINNING
---------------------------------

Logic Optimizing Compiler Utilization Report

 ***** Design implemented successfully

Lakshmi Jayanthi
DSO Applications
February 19, 1986


5C060


PART D: TWO DICE SPINNING

B Version 3.0, Baseline 17x, 9/26/85

                 5C060
               _ _ _ _ _
    clock1 -| 1    24|- Vcc
    switch -| 2    23|- GND
  RESERVED -| 3    22|- spin2d
  RESERVED -| 4    21|- spin2c
  RESERVED -| 5    20|- spin2b
  RESERVED -| 6    19|- spin2a
    spin1d -| 7    18|- RESERVED
    spin1c -| 8    17|- RESERVED
    spin1b -| 9    16|- RESERVED
    spin1a -|10    15|- RESERVED
       GND -|11    14|- GND
       GND -|12    13|- clock2
               _ _ _ _ _


**INPUTS**

                                                 |         Feeds:
    Name   Pin   Resource    MCell #      PTerms |   MCells    OE   Clear   Clock

   clock1    1        INP          -           -          -     -       -    CLK1

   switch    2        INP          -           -          1     -       -       -
                                                          2
                                                          3
                                                          4
                                                          5
                                                          6
                                                          7
                                                          8
                                                          9
                                                         10
                                                         11
                                                         12
                                                         13
                                                         14
                                                         15
                                                         16

   clock2   13        INP          -           -          -     -       -    CLK2
```

```
**OUTPUTS**

                                                     |          Feeds:
   Name   Pin   Resource    MCell #      PTerms      |   MCells       OE    Clear   Clock

 spin1d     7      RORF          13       2/ 8              13         -       -       -
                                                            14
                                                            15

 spin1c     8      RORF          14       2/ 8              13         -       -       -
                                                            14
                                                            15

 spin1b     9      RORF          15       2/ 8              13         -       -       -
                                                            14
                                                            15

 spin1a    10      RORF          16       1/ 8              13         -       -       -
                                                            14
                                                            15

 spin2a    19      RORF           4       1/ 8               1         -       -       -
                                                             2
                                                             3

 spin2b    20      RORF           3       2/ 8               1         -       -       -
                                                             2
                                                             3

 spin2c    21      RORF           2       2/ 8               1         -       -       -
                                                             2
                                                             3

 spin2d    22      RORF           1       2/ 8               1         -       -       -
                                                             2
                                                             3


**BURIED REGISTERS**

                                                     |          Feeds:
   Name   Pin   Resource    MCell #      PTerms      |   MCells       OE    Clear   Clock

           18      NORF           5       3/ 8               1         -       -       -
                                                             5
                                                             6
                                                             7
                                                             8

           17      NORF           6       4/ 8               2         -       -       -
                                                             5
                                                             6
                                                             7
                                                             8

           16      NORF           7       4/ 8               3         -       -       -
                                                             5
                                                             6
                                                             7
                                                             8
```

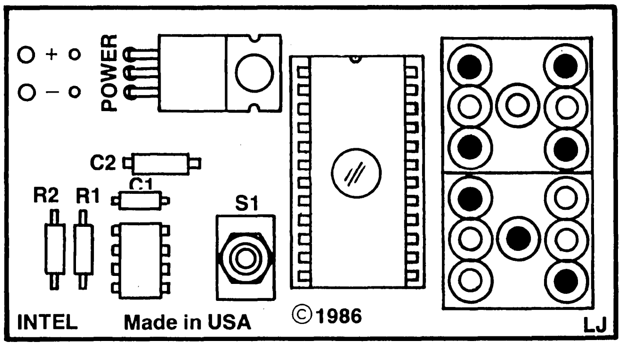
```
             15       NORF        8       7/ 8       4      -      -      -
                                                     5
                                                     6
                                                     7
                                                     8

              3       NORF        9       2/ 8       5      -      -      -
                                                     6
                                                     7
                                                     8
                                                     9
                                                    10
                                                    11
                                                    12
                                                    13

              4       NORF       10       3/ 8       9      -      -      -
                                                    10
                                                    11
                                                    12
                                                    14

              5       NORF       11       3/ 8       9      -      -      -
                                                    10
                                                    11
                                                    12
                                                    15

              6       NORF       12       5/ 8       9      -
                                                    10
                                                    11
                                                    12
                                                    16

**UNUSED RESOURCES**

        Name   Pin   Resource      MCell     PTerms

          -     11       -           -          -
          -     14       -           -          -
          -     23       -           -          -

**PART UTILIZATION**

86%       Pins
100%      MacroCells
35%       Pterms
```

NOTE: Part D of the design example utilizes the device in a very optimum manner. You have utilized all the macrocells and also 86% of the pins but only 35% of the product terms.

You have not used three of the input pins.

Consider this:

Make these three pins a mode select on this dice example — if all of these three additional inputs are high then the dice will function as described (this condition must be added to each product term). You now have seven other modes in which to operate this DICE. Anyone want to "load" the odds for "boxcars" or "snake-eyes"? You have 65% more product terms to use so you can be very creative. What else could you add to this EPLD?



Order NOW!

To order your PCB card and diskette with pre-entered design files, simply fill out the form below. Send a check, money order, or use your VISA or MasterCard, **or call toll-free 800-548-4725.**

Mail to: Intel Literature Sales, SC6-41
P.O. Box 58130
Santa Clara, CA 95052-8130

			Quantity	Price
555722	PCB card and diskette	$12.00 each	______	____________
Must add local sales tax				____________
		TOTAL		____________

Company ____

Name ________________________________ Title ________________

Address ____

City __________________________ State _________ Zip ____________

Business Phone ( ) ____________________________________

If paying by check or money order, please make payable to Intel Literature Sales.

☐ VISA ☐ MasterCard

Account Number ________________________________ Exp. Date _________

Source: CB (Allow 4-6 weeks for delivery)

LITERATURE

1987 HANDBOOKS (European Prices)

Product line handbooks contain data sheets, application notes, article reprints and other design information.

NAME	ORDER NUMBER	
MEMORY COMPONENTS HANDBOOK	210830	CATEGORY F
MICROCOMMUNICATIONS HANDBOOK	231658	CATEGORY F
EMBEDDED CONTROLLER HANDBOOK (includes Microcontrollers and 8085, 80186, 80188)	210918	CATEGORY F
MICROPROCESSOR AND PERIPHERAL HANDBOOK (2 Volume Set)	230843	CATEGORY H
DEVELOPMENT TOOLS HANDBOOK	210940	CATEGORY F
DOS DEVELOPMENT SOFTWARE CATALOG	280199	N/C
OEM BOARDS AND SYSTEMS HANDBOOK	280407	CATEGORY F
MILITARY HANDBOOK	210461	CATEGORY F
COMPONENTS QUALITY/RELIABILITY HANDBOOK	210997	CATEGORY F
SYSTEMS QUALITY/RELIABILITY HANDBOOK	231762	CATEGORY F
PRODUCT GUIDE Overview of Intel's complete product lines	210846	N/C
LITERATURE GUIDE List of Intel Literature	E00029	N/C
INTEL PACKAGING OUTLINES AND DIMENSIONS Packaging types, number of leads, etc.	231369	N/C

PRICE CODE	Finland FIM	Norway NKR	Sweden SEK	Denmark DKR	The Netherlands DFL	France FFR	U.K. PDS	Germany DM	Switzerland SFR	Austria AS
F	128.00	190.00	179.00	255.00	78.00	190.00	17.00	62.00	45.00	440.00
H	201.00	294.00	281.00	349.00	115.00	300.00	25.00	98.00	75.00	696.00

LOCAL TAX NOT INCLUDED

intel

EUROPEAN LITERATURE ORDER FORM

ORDER NUMBER	TITLE	QTY		PRICE	TOTAL
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		
______ - ___			X		

SUB TOTAL ____________

LOCAL TAX ____________

TOTAL ____________

PAYMENT

Cheques should be made payable to your local Intel Sales Office (see inside back cover).

Other forms of payment may be available in your country. Please contact the Literature Co-ordinator at your local Intel Sales Office for details.

The Completed form should be marked for the attention of the LITERATURE CO-ORDINATOR and returned to your local Intel Sales Office.

NAME ____

COMPANY ____

ADDRESS ____

____

____

____________________ ____________________

PHONE NO. ____

intel®

DOMESTIC SALES OFFICES

ALABAMA

Intel Corp.
5015 Bradford Drive
Suite 2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Drive
Suite 214D
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 540-6040

Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

Intel Corp.
4350 Executive Drive
Suite 105
San Diego 92121
Tel: (619) 452-5880

Intel Corp.*
2000 East 4th Street
Suite 100
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
Santa Clara, CA 95051
Tel: (408) 986-8086
TWX: 910-338-0255

COLORADO

Intel Corp.
3300 Mitchell Lane, Suite 210
Boulder 80301
Tel: (303) 442-8088

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (303) 594-6622

Intel Corp.*
650 S. Cherry Street
Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
Danbury 06810
Tel: (203) 748-3130
TWX: 710-456-1199

EMC Corp.
222 Summer Street
Stamford 06901
Tel: (203) 327-2934

FLORIDA

Intel Corp.
242 N. Westmonte Drive
Suite 105
Altamonte Springs 32714
Tel: (305) 869-5588

Intel Corp.
6363 N.W. 6th Way, Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

FLORIDA (Cont'd)

Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33702
Tel: (813) 577-2413

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

Intel Corp.*
300 N. Martingale Road, Suite 400
Schaumburg 60172
Tel: (312) 310-8031

INDIANA

Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Intel Corp.
7833 Walker Drive
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

Intel Corp.*
Westford Corp. Center
3 Carlisle Road
Westford 01886
Tel: (617) 692-3222
TWX: 710-343-6333

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8096

MINNESOTA

Intel Corp.
3500 W. 80th Street
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

Intel Corp.
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111

NEW MEXICO

Intel Corp.
8500 Menual Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.*
300 Vanderbilt Motor Parkway
Hauppauge 11788
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
Suite 2B Hollowbrook Park
15 Myers Corners Road
Wappinger Falls 12590
Tel: (914) 297-6161
TWX: 510-248-0060

Intel Corp.*
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

NORTH CAROLINA

Intel Corp.
5700 Executive Center Drive
Suite 213
Charlotte 28212
Tel: (704) 568-8966

Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
25700 Science Park Drive
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73116
Tel: (405) 848-8086

OREGON

Intel Corp.
15254 N.W. Greenbrier Parkway, Bldg. B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

PENNSYLVANIA

Intel Corp.
1513 Cedar Cliff Drive
Camphill 17011
Tel: (717) 737-5035

Intel Corp.*
455 Pennsylvania Avenue
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Boulevard
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-3030

TEXAS

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

Intel Corp.*
12300 Ford Road
Suite 380
Dallas 75234
Tel: (214) 241-8087
TWX: 910-860-5617

Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

Industrial Digital Systems Corp.
5925 Sovereign
Suite 101
Houston 77036
Tel: (713) 988-9421

UTAH

Intel Corp.
5201 Green Street
Suite 290
Murray 84123
Tel: (801) 263-8051

VIRGINIA

Intel Corp.
1603 Santa Rosa Road
Suite 109
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
155-108 Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

Intel Corp.
450 N. Sunnyslope Road
Suite 130
Chancellory Park 1
Brookfield 53005
Tel: (414) 784-8087

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
301-2245 W. Broadway
Vancouver V6K 2E4
Tel: (604) 738-6522

ONTARIO

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TELEX: 053-4115

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TELEX: 06983574

QUEBEC

Intel Semiconductor of Canada, Ltd.
620 St. Jean Boulevard
Pointe Claire H9R 3K3
Tel: (514) 694-9130
TWX: 514-694-9134

*Field Application Location

intel®

DOMESTIC DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc.
1015 Henderson Road
Huntsville 35805
Tel: (205) 837-6955

†Hamilton/Avnet Electronics
4812 Commercial Drive N.W.
Huntsville 35805
Tel: (205) 837-7210
TWX: 810-726-2162

Pioneer/Technologies Group Inc.
4825 University Square
Huntsville 35805
Tel: (205) 837-9300
TWX: 810-726-2197

ARIZONA

†Hamilton/Avnet Electronics
505 S. Madison Drive
Tempe 85281
Tel: (602) 231-5100
TWX: 910-950-0077

Kierulff Electronics
4134 E. Wood Street
Phoenix 85040
Tel: (602) 437-0750
TWX: 910-951-1550

Wyle Distribution Group
17855 N. Black Canyon Highway
Phoenix 85023
Tel: (602) 866-2888

CALIFORNIA

Arrow Electronics, Inc.
19748 Dearborn Street
Chatsworth 91311
Tel: (818) 701-7500
TWX: 910-493-2086

Arrow Electronics, Inc.
1502 Crocker Avenue
Hayward 94544
Tel: (408) 487-4600

Arrow Electronics
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
TLX: 888064

†Arrow Electronics, Inc.
521 Weddell Drive
Sunnyvale 94086
Tel: (408) 745-6600
TWX: 910-339-9371

Arrow Electronics, Inc.
2961 Dow Avenue
Tustin 92680
Tel: (714) 838-5422
TWX: 910-595-2860

†Avnet Electronics
350 McCormick Avenue
Costa Mesa 92626
Tel: (714) 754-6051
TWX: 910-595-1928

Hamilton/Avnet Electronics
1175 Bordeaux Drive
Sunnyvale 94086
Tel: (408) 743-3300
TWX: 910-339-9332

†Hamilton/Avnet Electronics
4545 Viewridge Avenue
San Diego 92123
Tel: (619) 571-7500
TWX: 910-595-2638

†Hamilton/Avnet Electronics
20501 Plummer Street
Chatsworth 91311
Tel: (818) 700-6271
TWX: 910-494-2207

†Hamilton/Avnet Electronics
4103 Northgate Boulevard
Sacramento 95834
Tel: (916) 920-3150

Hamilton/Avnet Electronics
3002 G Street
Ontario 91311
Tel: (714) 989-9411

Hamilton/Avnet Electronics
19515 So. Vermont Avenue
Torrance 90502
Tel: (213) 615-3909
TWX: 910-349-6263

Hamilton Electro Sales
9650 De Soto Avenue
Chatsworth 91311
Tel: (818) 700-6500

†Hamilton Electro Sales
10950 W. Washington Boulevard
Culver City 90230
Tel: (213) 558-2458
TWX: 910-340-6364

CALIFORNIA (Cont'd)

Hamilton Electro Sales
1361 B West 190th Street
Gardena 90248
Tel: (213) 558-2131

†Hamilton Electro Sales
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4150
TWX: 910-595-2638

Kierulff Electronics
10824 Hope Street
Cypress 90430
Tel: (714) 220-6300

Kierulff Electronics, Inc.
1180 Murphy Avenue
San Jose 95131
Tel: (408) 971-2600
TWX: 910-379-6430

Kierulff Electronics, Inc.
14101 Franklin Avenue
Tustin 92680
Tel: (714) 731-5711
TWX: 910-595-2599

†Kierulff Electronics, Inc.
5650 Jillson Street
Commerce 90040
Tel: (213) 725-0325
TWX: 910-580-3666

Wyle Distribution Group
26560 Agoura Street
Calabasas 91302
Tel: (818) 880-9000
TWX: 818-372-0232

†Wyle Distribution Group
124 Maryland Street
El Segundo 90245
Tel: (213) 322-8100
TWX: 910-348-7140 or 7111

†Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel: (714) 843-9953
TWX: 910-595-1572

†Wyle Distribution Group
11151 Sun Center Drive
Rancho Cordova 95670
Tel: (916) 638-5282

†Wyle Distribution Group
9525 Chesapeake Drive
San Diego 92123
Tel: (619) 565-9171
TWX: 910-335-1590

†Wyle Distribution Group
3000 Bowers Avenue
Santa Clara 95051
Tel: (408) 727-2500
TWX: 910-338-0296

Wyle Military
18910 Teller Avenue
Irvine 92750
Tel: (714) 851-9958
TWX: 310-371-9127

Wyle Systems
7382 Lampson Avenue
Garden Grove 92641
Tel: (714) 851-9953
TWX: 910-595-2642

COLORADO

Arrow Electronics, Inc.
1390 S. Potomac Street
Suite 136
Aurora 80012
Tel: (303) 696-1111

†Hamilton/Avnet Electronics
8765 E. Orchard Road
Suite 708
Englewood 80111
Tel: (303) 740-1017
TWX: 910-935-0787

†Wyle Distribution Group
451 E. 124th Avenue
Thornton 80241
Tel: (303) 457-9953
TWX: 910-936-0770

CONNECTICUT

†Arrow Electronics, Inc.
12 Beaumont Road
Wallingford 06492
Tel: (203) 265-7741
TWX: 710-476-0162

†Hamilton/Avnet Electronics
Commerce Industrial Park
Commerce Drive
Danbury 06810
Tel: (203) 797-2800
TWX: 710-456-9974

CONNECTICUT (Cont'd)

†Pioneer Northeast Electronics
112 Main Street
Norwalk 06851
Tel: (203) 853-1515
TWX: 710-468-3373

FLORIDA

†Arrow Electronics, Inc.
350 Fairway Drive
Deerfield Beach 33441
Tel: (305) 429-8200
TWX: 510-955-9456

†Arrow Electronics, Inc.
1001 N.W. 62nd Street
Suite 108
Ft. Lauderdale 33309
Tel: (305) 776-7790
TWX: 510-955-9456

†Arrow Electronics, Inc.
50 Woodlake Drive W., Bldg. B
Palm Bay 32905
Tel: (305) 725-1480
TWX: 510-959-6337

†Hamilton/Avnet Electronics
6801 N.W. 15th Way
Ft. Lauderdale 33309
Tel: (305) 971-2900
TWX: 510-956-3097

†Hamilton/Avnet Electronics
3197 Tech Drive North
St. Petersburg 33702
Tel: (813) 576-3930
TWX: 810-863-0374

Hamilton/Avnet Electronics
6947 University Boulevard
Winterpark 32792
Tel: (305) 628-3888
TWX: 810-853-0322

†Pioneer Electronics
221 N. Lake Boulevard
Suite 412
Alta Monte Springs 32701
Tel: (305) 834-9090
TWX: 810-853-0284

†Pioneer Electronics
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
TWX: 510-955-9653

GEORGIA

†Arrow Electronics, Inc.
3155 Northwoods Parkway, Suite A
Norcross 30071
Tel: (404) 449-8252
TWX: 810-766-0439

Hamilton/Avnet Electronics
5825 D. Peachtree Corners
Norcross 30092
Tel: (404) 447-7500
TWX: 810-766-0432

Pioneer Electronics
5835B Peachtree Corners E
Norcross 30092
Tel: (404) 448-1711
TWX: 810-766-4515

ILLINOIS

†Arrow Electronics, Inc.
2000 E. Alonquin Street
Schaumberg 60195
Tel: (312) 397-3440
TWX: 910-291-3544

†Hamilton/Avnet Electronics
1130 Thorndale Avenue
Bensenville 60106
Tel: (312) 860-7780
TWX: 910-227-0060

MTI Systems Sales
1100 West Thorndale
Itasca 60143
Tel: (312) 773-2300

†Pioneer Electronics
1551 Carmen Drive
Elk Grove Village 60007
Tel: (312) 437-9680
TWX: 910-222-1834

INDIANA

†Arrow Electronics, Inc.
2495 Directors Row, Suite H
Indianapolis 46241
Tel: (317) 243-9353
TWX: 810-341-3119

Hamilton/Avnet Electronics
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
TWX: 810-260-3966

INDIANA (Cont'd)

†Pioneer Electronics
6408 Castleplace Drive
Indianapolis 46250
Tel: (317) 849-7300
TWX: 810-260-1794

KANSAS

†Hamilton/Avnet Electronics
9219 Quivera Road
Overland Park 66215
Tel: (913) 888-8900
TWX: 910-743-0005

KENTUCKY

Hamilton/Avnet Electronics
1051 D. Newton Park
Lexington 40511

MARYLAND

Arrow Electronics, Inc.
8300 Gulford Road #H
Rivers Center
Columbia 21046
Tel: (301) 995-0003
TWX: 710-236-9005

†Hamilton/Avnet Electronics
6822 Oak Hall Lane
Columbia 21045
Tel: (301) 995-3500
TWX: 710-862-1861

†Mesa Technology Corporation
16021 Industrial Drive
Gaithersburg 20877
Tel: (301) 948-4350
Twx: 710-828-9702

†Pioneer Electronics
9100 Gaither Road
Gaithersburg 20877
Tel: (301) 948-0710
TWX: 710-828-0545

MASSACHUSETTS

†Arrow Electronics, Inc.
1 Arrow Drive
Woburn 01801
Tel: (617) 933-8130
TWX: 710-393-6770

†Hamilton/Avnet Electronics
10D Centennial Drive
Peabody 01960
Tel: (617) 532-3701
TWX: 710-393-0382

MTI Systems Sales
13 Fortune Drive
Billerica 01821

Pioneer Northeast Electronics
44 Hartwell Avenue
Lexington 02173
Tel: (617) 863-1200
TWX: 710-326-6617

MICHIGAN

Arrow Electronics, Inc.
755 Phoenix Drive
Ann Arbor 48104
Tel: (313) 971-8220
TWX: 810-223-6020

†Hamilton/Avnet Electronics
32487 Schoolcraft Road
Livonia 48150
Tel: (313) 522-4700
TWX: 810-242-8775

Hamilton/Avnet Electronics
2215 29th Street S.E.
Space A5
Grand Rapids 49508
Tel: (616) 243-8805
TWX: 810-273-6921

†Pioneer Electronics
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
TWX: 810-242-3271

MINNESOTA

†Arrow Electronics, Inc.
5230 W. 73rd Street
Edina 55435
Tel: (612) 830-1800
TWX: 910-576-3125

Hamilton/Avnet Electronics
10300 Bren Road East
Minnetonka 55343
Tel: (612) 932-0600
TWX: (910) 576-2720

†Pioneer Electronics
10203 Bren Road East
Minnetonka 55343
Tel: (612) 935-5444
TWX: 910-576-2738

†Microcomputer System Technical Demonstrator Centers

CG-11/6/86

intel®

DOMESTIC DISTRIBUTORS

MISSOURI

†Arrow Electronics, Inc.
2380 Schuetz
St. Louis 63141
Tel: (314) 567-6888
TWX: 910-764-0882

†Hamilton/Avnet Electronics
13743 Shoreline Court
Earth City 63045
Tel: (314) 344-1200
TWX: 910-762-0684

NEW HAMPSHIRE

†Arrow Electronics, Inc.
3 Perimeter Road
Manchester 03103
Tel: (603) 668-6968
TWX: 710-220-1684

Hamilton/Avnet Electronics
444 E. Industrial Drive
Manchester 03104
Tel: (603) 624-9400

NEW JERSEY

†Arrow Electronics, Inc.
6000 Lincoln East
Marlton 08053
Tel: (609) 596-8000
TWX: 710-897-0829

†Arrow Electronics, Inc.
2 Industrial Road
Fairfield 07006
Tel: (201) 575-5300
TWX: 710-998-2206

†Hamilton/Avnet Electronics
1 Keystone Avenue
Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
TWX: 710-940-0262

†Hamilton/Avnet Electronics
10 Industrial
Fairfield 07006
Tel: (201) 575-3390
TWX: 701-734-4388

†Pioneer Northeast Electronics
45 Route 46
Pinebrook 07058
Tel: (201) 575-3510
TWX: 710-734-4382

†MTI Systems Sales
383 Route 46 W
Fairfield 07006
Tel: (201) 227-5552

NEW MEXICO

Alliance Electronics Inc.
11030 Cochiti S.E.
Albuquerque 87123
Tel: (505) 292-3360
TWX: 910-989-1151

Hamilton/Avnet Electronics
2524 Baylor Drive S.E.
Albuquerque 87106
Tel: (505) 765-1500
TWX: 910-989-0614

NEW YORK

†Arrow Electronics, Inc.
25 Hub Drive
Melville 11747
Tel: (516) 694-6800
TWX: 510-224-6126

†Arrow Electronics, Inc.
3375 Brighton-Henrietta Townline Road
Rochester 14623
Tel: (716) 427-0300
TWX: 510-253-4766

Arrow Electronics, Inc.
7705 Maltage Drive
Liverpool 13088
Tel: (315) 652-1000
TWX: 710-545-0230

Arrow Electronics, Inc.
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
TWX: 510-227-6623

Hamilton/Avnet Electronics
333 Metro Park
Rochester 14623
Tel: (716) 475-9130
TWX: 510-253-5470

Hamilton/Avnet Electronics
103 Twin Oaks Drive
Syracuse 13206
Tel: (315) 437-2641
TWX: 710-541-1560

NEW YORK (Cont'd)

†Hamilton/Avnet Electronics
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
TWX: 510-224-6166

†MTI Systems Sales
38 Harbor Park Drive
P.O. Box 271
Port Washington 11050
Tel: (516) 621-6200
TWX: 510-223-0846

†Pioneer Northeast Electronics
1806 Vestal Parkway East
Vestal 13850
Tel: (607) 748-8211
TWX: 510-252-0893

†Pioneer Northeast Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
TWX: 510-221-2184

Pioneer Northeast Electronics
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
TWX: 510-253-7001

NORTH CAROLINA

Arrow Electronics, Inc.
5240 Greendairy Road
Raleigh 27604
Tel: (919) 876-3132
TWX: 510-928-1856

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819
TWX: 510-928-1836

Pioneer Electronics
9801 A-Southern Pine Boulevard
Charlotte 28210
Tel: (704) 524-8188
TWX: 810-621-0366

OHIO

Arrow Electronics, Inc.
7620 McEwen Road
Centerville 45459
Tel: (513) 435-5563
TWX: 810-459-1611

†Arrow Electronics, Inc.
6238 Cochran Road
Solon 44139
Tel (216) 248-3990
TWX: 810-427-9409

†Hamilton/Avnet Electronics
954 Senate Drive
Dayton 45459
Tel: (513) 433-0610
TWX: 810-450-2531

†Hamilton/Avnet Electronics
4588 Emery Industrial Parkway
Warrensville Heights 44128
Tel: (216) 831-3500
TWX: 810-427-9452

†Pioneer Electronics
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
TWX: 810-459-1622

†Pioneer Electronics
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
TWX: 810-422-2211

OKLAHOMA

Arrow Electronics, Inc.
4719 S. Memorial Drive
Tulsa 74145
Tel: (918) 665-7700

OREGON

†Almac Electronics Corporation
1885 N.W. 169th Place
Beaverton 97006
Tel: (503) 629-8090
TWX: 910-467-8746

Hamilton/Avnet Electronics
6024 S.W. Jean Road
Bldg. C, Suite 10
Lake Oswego 97034
Tel: (503) 635-7848
TWX: 910-455-8179

Wyle Distribution Group
5250 N.E. Elam Young Parkway
Suite 600
Hillsboro 97124
Tel: (503) 640-6000
TWX: 910-460-2203

PENNSYLVANIA

Arrow Electronics, Inc.
650 Seco Road
Monroeville 15146
Tel: (412) 856-7000

Pioneer Electronics
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
TWX: 710-795-3122

†Pioneer Electronics
261 Gibralter Road
Horsham 19044
Tel: (215) 674-4000
TWX: 510-665-6778

TEXAS

†Arrow Electronics, Inc.
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
TWX: 910-860-5377

†Arrow Electronics, Inc.
10899 Kinghurst
Suite 100
Houston 77099
Tel: (713) 530-4700
TWX: 910-880-4439

Arrow Electronics, Inc.
10125 Metropolitan
Austin 78758
Tel: (512) 835-4180
TWX: 910-874-1348

†Hamilton/Avnet Electronics
1807 W. Braker Lane
Austin 78758
Tel: (512) 837-8911
TWX: 910-874-1319

†Hamilton/Avnet Electronics
2111 W. Walnut Hill Lane
Irving 75062
Tel: (214) 659-4100
TWX: 910-860-5929

†Hamilton/Avnet Electronics
4850 Wright Road #190
Houston 77477
Tel: (713) 780-1771
TWX: 910-881-5523

†Pioneer Electronics
9901 Burnet Road
Austin 78758
Tel: (512) 835-4000
TWX: 910-874-1323

Pioneer Electronics
13710 Omega Road
Dallas 75234
Tel: (214) 386-7300
TWX: 910-850-5563

Pioneer Electronics
5853 Point West Drive
Houston 77036
Tel: (713) 988-5555
TWX: 910-881-1606

UTAH

†Hamilton/Avnet Electronics
1585 West 2100 South
Salt Lake City 84119
Tel: (801) 972-2800
TWX: 910-925-4018

Wyle Distribution Group
1959 South 4130 West, Unit B
Salt Lake City 84104
Tel: (801) 974-9953

WASHINGTON

†Almac Electronics Corporation
14360 S.E. Eastgate Way
Bellevue 98007
Tel: (206) 643-9992
TWX: 910-444-2067

Arrow Electronics, Inc.
14320 N.E. 21st Street
Bellevue 98007
Tel: (206) 643-4800
TWX: 910-444-2017

Hamilton/Avnet Electronics
14212 N.E. 21st Street
Bellevue 98005
Tel: (206) 453-5874
TWX: 910-443-2469

WISCONSIN

†Arrow Electronics, Inc.
430 W. Rausson Avenue
Oakcreek 53154
Tel: (414) 764-6600
TWX: 910-262-1193

WISCONSIN (Cont'd)

Hamilton/Avnet Electronics
2975 Moorland Road
New Berlin 53151
Tel: (414) 784-4510
TWX: 910-262-1182

CANADA

ALBERTA

Hamilton/Avnet Electronics
2816 21st Street N.E.
Calgary T2E 6Z2
Tel: (403) 230-3586
TWX: 03-827-642

Hamilton/Avnet Electronics
6845 Rexwood Road Unit 6
Mississauga, Ontario L4V1R2
Tel: (416) 677-0484

Zentronics
Bay No. 1
3300 14th Avenue N.E.
Calgary T2A 6J4
Tel: (403) 272-1021

BRITISH COLUMBIA

Hamilton/Avnet Electronics
105-2550 Boundry Road
Burmalay V5M 3Z3
Tel: (604) 272-4242

Zentronics
108-11400 Bridgeport Road
Richmond V6X 1T2
Tel: (604) 273-5575
TWX: 04-5077-89

MANITOBA

Zentronics
590 Berry Street
Winnipeg R3H OS1
Tel: (204) 775-8661

ONTARIO

Arrow Electronics Inc.
24 Martin Ross Avenue
Downsview M3J 2K9
Tel: (416) 661-0220
TELEX: 06-218213

Arrow Electronics Inc.
148 Colonnade Road
Nepean K2E 7J5
Tel: (613) 226-6903

†Hamilton/Avnet Electronics
6845 Rexwood Road
Units G & H
Mississauga L4V 1R2
Tel: (416) 677-7432
TWX: 610-492-8867

†Hamilton/Avnet Electronics
210 Colonnade Road South
Nepean K2E 7L5
Tel: (613) 226-1700
TWX: 05-349-71

†Zentronics
8 Tilbury Court
Brampton L6T 3T4
Tel: (416) 451-9600
TWX: 06-976-78

Zentronics
564/10 Weber Street North
Waterloo N2L 5C6
Tel: (519) 884-5700

Zentronics
155 Colonnade Road
Unit 17
Nepean K2E 7K1
Tel: (613) 225-8840
TWX: 06-976-78

QUEBEC

Arrow Electronics Inc.
4050 Jean Talon Quest
Montreal H4P 1W1
Tel: (514) 735-5511
TELEX: 05-25596

Arrow Electronics Inc.
909 Charest Blvd.
Quebec 61N 269
Tel: (418) 687-4231
TLX: 05-13388

Hamilton/Avnet Electronics
2795 Rue Halpern
St. Laurent H4S 1P8
Tel: (514) 335-1000
TWX: 610-421-3731

Zentronics
505 Locke Street
St. Laurent H4T 1X7
Tel: (514) 735-5361
TWX: 05-827-535

†Microcomputer System Technical Demonstrator Centers

EUROPEAN SALES OFFICES

BELGIUM

Intel Corporation S.A.
Parc Seny
Rue du Moulin a Papier 51
Boite 1
B-1160 Brussels
Tel: (02) 661 07 11
TELEX: 24814

DENMARK

Intel Denmark A/S*
Glentevej 61 - 3rd Floor
DK-2400 Copenhagen
Tel: (01) 19 80 33
TELEX: 19567

FINLAND

Intel Finland OY
Ruosilantie 2
000390 Helsinki
Tel: (0) 544 644
TELEX: 123 332

FRANCE

Intel Paris
1 Rue Edison, BP 303
78054 Saint-Quentin en Yvelines
Tel: (33) 1 30 57 70 00
TELEX: 69901677

FRANCE (Cont'd)

Intel Corporation, S.A.R.L.
Immeuble BBC
4 Quai des Etroits
69005 Lyon
Tel: (7) 842 40 89
TELEX: 305153

WEST GERMANY

Intel Semiconductor GmbH*
Seidlstrasse 27
D-8000 Munchen 2
Tel: (89) 53891
TELEX: 05-23177 INTL D

Intel Semiconductor GmbH*
Mainzerstrasse 75
D-6200 Wiesbaden 1
Tel: (6121) 70 08 74
TELEX: 04168183 INTW D

Intel Semiconductor GmbH
Bruckstrasse 61
7012 Fellbach
Stuttgart
Tel: (711) 58 00 82
TELEX: 7254826 INTS D

Intel Semiconductor GmbH*
Hohenzollernstrasse 5*
3000 Hannover 1
Tel: (511) 34 40 81
TELEX: 923625 INTH D

ISRAEL

Intel Semiconductors Ltd*
Atidim Industrial Park
Neve Sharet
Dvora Hanevia
Bldg. No. 13, 4th Floor
P.O. Box 43202
Tel Aviv 61430
Tel: 3-491099/8
TELEX: 371215

ITALY

Intel Corporation Italia Spa*
Milanofiori, Palazzo E/3
20094 Assago (Milano)
Tel: (02) 824 40 71
TELEX: 315183 INTMIL

NETHERLANDS

Intel Semiconductor Nederland B.V.*
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam
Tel: (10) 21 23 77
TELEX: 22283

NORWAY

Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013
Skjetten
Tel: (2) 742 420
TELEX: 18018

SPAIN

Intel Iberia
Calle Zurbaran
28-1-IZQDA
28010 Madrid
Tel: (34) 1410 40 04
TELEX: 46880

SWEDEN

Intel Sweden A.B.*
Dalvagen 24
S-171 36 Solna
Tel: 8/7340100
TELEX: 12261

SWITZERLAND

Intel Semiconductor A.G.*
Talackerstrasse 17
8152 Glattbrugg postfach
CH-8065 Zurich
Tel: (01) 829 29 77
TELEX: 57989 ICH CH

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.*
Pipers Way
Swindon, Wiltshire SN3 1RJ
Tel: (793) 696 000
TELEX: 444447 INT SWN

*Field Application Location

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Elektronische Ges.m.b.H.
Meidlinger Hauptstrasse 78
A 1120 Wien
Tel: (222) 83 56 46
TELEX: 11532 BASAT A

Moor Ges.m.b.H.
Storchengasse 1/1/1
A-1150 Wien
Tel: 222-85 86 46

BELGIUM

S.A. Inelco Belgium
Ave. des Croix de Guerre 94
B1120 Brussels
Tel: (02) 216 01 60
TELEX: 25441

ITT Electronic Components
rue Colonel Bourgstr. 105a (Bte 3)
B-1140 Brussels
Tel: (02) 735-7125

DENMARK

ITT MultiKomponent A/S
Naverland 29
DK-2600 Glostrup
Tel: (02) 456 66 45
TX: 33355

FINLAND

Oy Fintronic AB
Melkonkatu 24 A
SF-00210 Helsinki 21
Tel: (0) 692 60 22
TELEX: 124 224 Ftron SF

FRANCE

Generim
Z.A. de Courtaboeuf
Avenue de la Baltique
F-91943 Les Ulis Cedex-B.P. 88
Tel: (1) 69 07 78 78
TELEX: F691700

Jermyn
16, Avenue de Jean-Jaures
94600 Choisy-Le-Roi
Tel: (1) 48 53 12 00
TELEX: 260 967

Metrologie
Tour d' Asnieres
4, Avenue Laurent Cely
92606-Asnieres
Tel: (1) 47 90 62 40
TELEX: 611-448

Tekelec Airtronic
Cite des Bruyeres
Rue Carle Vernet B.P. 2
92310 Sevre S
Tel: (1) 45 34 75 35
TELEX: 204552

WEST GERMANY

Electronic 2000 Vertriebs A.G.
Stahlgruberring 12
8000 Munich 82
Tel: (89) 42 00 10
TELEX: 522561 EEC D

Jermyn GmbH
Postfach 11 80
Schulstrasse 84
6277 Bad Camberg
Tel: (06434) 231
TELEX: 484426 JERM D

CES Computer Electronics Systems GmbH
AM Moosfeld 37
8000 Munich 82
Tel: (089) 420-430
TELEX: 2180260

Metrologie GmbH
Hansastrasse 15
8000 Munich 21
Tel: (89) 570-940
TELEX: 5213189

Proelectron Vertriebs AG
Max Planck Strasse 1-3
6072 Dreieich
Tel: (6103) 3040
TELEX: 417983

ITT-Multikomponent
Bahnhofstrasse 44
7141 Moeglingen
Tel: (07141) 4870

IRELAND

Micro Marketing
Glenageary Office Park
Glenageary
Co. Dublin
Tel: (1) 85 62 88
TELEX: 31584

ISRAEL

Eastronics Ltd.
11 Rozanis Street
P.O. Box 39300
Tel Aviv 61392
Tel: (3) 47 51 51
TELEX: 33638

ITALY

Eledtra 3S S.P.A.
Via G. Watt, 37
I 20143 Milano
Tel: (2) 81 82 1
TELEX: 332332

ITALY (Cont'd)

Intesi
Milanofiori E/5
20090 Assago
Tel: (2) 824701
TELEX: 311351

Lasi Elettronica S.P.A.
V. Le Fulvio Testi, 126
20092 Cinisello Balsamo (Milano)
Tel: 02/2440012
TELEX: 352040 LASIMI I
TELEFAX: 2487717

NETHERLANDS

Koning & HartmanElectrotechniek B.V.
P.O. Box 125
2600 AC Delft
Tel: 15 609-906
TELEX: 31528

NORWAY

Nordisk Elektronic A/S
Post Office Box 130
N-1364 Hvalstad
Tel: (2) 846 210
TELEX: 17546

PORTUGAL

Ditram
Avenida Miguel Bombarda, 133
P-1000 Lisboa
Tel: (1) 154 53 13
TELEX: 14182 Brieks-P

SPAIN

ITT SESA
Miguel Angel 21-3
Madrid 28010
Tel: (1) 419 54 00
TELEX: 27461

Diode
Calle Gandesa 10-4
08028 Barcelona
Tel: (3) 322-12-51
TELEX: 42148

SWEDEN

Nordisk Electronik AB
Huvudstagatan 1
Box 1409
S-171 27 Solna
Tel: (8) 734 97 70
TELEX: 10547

SWITZERLAND

Industrade AG
Hertistrasse
CH-8304 Wallisellen
Tel: (01) 830 50 40
TELEX: 56788 INDEL CH

UNITED KINGDOM

Accent Electronic Components Ltd.
Jubilee House
Jubilee Way
Letchworth
Hertfordshire SG6 1QH
Tel: (0462) 686666
TELEX: 826293

Bytech Ltd.
2 The Western Center
Western Industrial Estate
Bracknell
Berkshire RG12 1RW
Tel: (0344) 482211
TELEX: 848215

Comway Microsystems Ltd.
Market Street
Bracknell
Berkshire
Tel: (344) 55333
TELEX: 847201

IBR Microcomputers Ltd.
Unit 2 Western Center
Western Industrial Estate
Bracknell
Berkshire RG12 1RW
Tel: (0344) 486555
TELEX: 849381

Jermyn Industries Vestry Estate
Oxford Road
Seven Oaks
Kent TN 14 5EU
Tel: (0732) 450144
TELEX: 95142

M.E.D.L.
East Lane Road
North Wembley
Middlesex HA9 7PP
Tel: (190) 49307
TELEX: 28817

Rapid Recall, Ltd.
Rapid House/Denmark St.
High Wycombe
Bucks HP11 2ER
Tel: (494) 26 271
TELEX: 837931

IBR
2 The Western Center
Western Road
Bracknell, Berkshire
Tel: (0344) 486-555

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.*
Spectrum Building
200 Pacific Highway
Level 6
Crows Nest, NSW, 2065
Tel: 011-61-2-957-2744
TELEX: 970-20097
FAX: 011-61-2-957-2744

CHINA

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Avenue
Beijing, PRC

HONG KONG

Intel Semiconductor Ltd.*
1701-3 Connaught Centre
1 Connaught Road
Tel: 011-852-5-215-311
TWX: 60410 ITLHK

JAPAN

Intel Japan K.K.
5-6 Tokodai, Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Tel: 029747-8511
TELEX: 03656-160

Intel Japan K.K.*
Komeshin Bldg.
2-1-15 Naka-machi
Atsugi, Kanagawa 243
Tel: 0462-23-3511

Intel Japan K.K.*
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183
Tel: 0423-60-7871

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya, Saitama 360
Tel: 0485-24-6871

Intel Japan K.K.*
Ryokuchi-Station Bldg.
2-4-1 Terauchi
Toyonaka, Osaka 560
Tel: 06-863-1091

JAPAN (Cont'd)

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-201-3621

Intel Japan K.K.*
Flower-Hill Shin-machi East Bldg.
1-23-9 Shinmachi
Setagaya-ku, Tokyo 154
Tel: 03-426-2231

Intel Japan K.K.*
Mitsui-Seimei Musashi-Kosugi Bldg.
915-20 Shinmaruko, Nakahara-ku
Kawasaki-Shi, Kanagawa 211
Tel: 044-733-7011

Intel Japan K.K.
Mishima Tokyo-Kaijo Bldg.
1-1 Shibahon-cho
Mishima-shi
Shizuoka-Ken 411
Tel: 0559-72-4121

KOREA

Intel Semiconductor Asia Ltd.
Singsong Bldg. 8th Floor #906
25-4 Yoido-Dong, Youngdeungpo-Ku
Seoul 150
Tel: 011-82-2-784-8186 or 8286
TELEX: K29312 INTELKO

SINGAPORE

Intel Semiconductor Ltd.
101 Thomson Road
21-06 Goldhill Square
Singapore 1130
Tel: 011-65-250-7811
TWX: RS 39921

TAIWAN

Intel Semiconductor Ltd.
Rm 808, Min Chi Bldg.
746 Min Sheng East Road
Taipei
Tel: 011-886-2-716-9660

*Field Application Location

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

VLC S.R.L. Bartalome Mitre 1711
3 Piso
1037 Buenos Aires
Tel: 011-54-1-49-2092
TELEX: 17575 EDARG-AR

AUSTRALIA

Total Electronics
(Mailing Address)
Private Bag 250
Burwood, Victoria 3125

(Shipping Address)
9 Harker Street
Burwood
Victoria 3125
Tel: 011-61-3-288-4044
TELEX: AA 31261

Total Electronics
P.O. Box 139
Artamon, N.S.W. 2064
Tel: 011-61-02-438-1855
TELEX: 26297

BRAZIL

Elebra Microelectronica S/A
R. Florida, 1821-8 ander
04571 - Sao Paulo-SP
Tel: 011-55-11-533-9977
TELEX: 1125957

CHILE

DIN Instruments
Casilla 6055, Correo 22
Santiago
Tel: 225-8139
TELEX: 440422 Rudy CZ

(Shipping Address)
A102 Greenville Center
3801 Kennett Pike
Wilmington, Delaware 19807

CHINA

Novel Precision Machinery Co., Ltd.
Flat D 20 Kingsford Ind. Bldg.
Phase 1 26 Kwai Hei Street NT
Hong Kong
Tel: 011-852-5-223222
TWX: 39114 JINMI HK

CHINA (Cont'd)

Schmidt & Co. Ltd.
18/F. Great Eagle Centre
Wanchai
Hong Kong
Tel: 011-852-5-822-0222
TWX: 74766 SCHMC HK

HONG KONG

Schmidt & Co. Ltd.
18/F. Great Eagle Centre
Wanchai
Tel: 011-852-5-822-0222
TWX: 74766 SCHMC HK

INDIA

Micronic Devices
65 Arun Complex
D V G Road
Basavan Gudi
Bangalore 560 004
Tel: 011-91-812-600-631
TELEX: 011-5947 MDEV

Micronic Devices
104/109C Nirmal Industrial Estate
Sion (E)
Bombay 400 022
Tel: 011-91-22-48-61-70
TELEX: 011-71447 MDEV IN

Micronic Devices
R-694 New Rajinder Nager
New Delhi 110 060

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. Room 407
2-14-1 Asano, Kokurakita-Ku
Kitakyushu City 802
Tel: (093) 511-6471
TELEX: AECKY 7126-16

C. Itoh Micronics Corp.
OS 85 Bldg. 2-6-5 Suda-Cho
Kanda Chiyoda-Ku, Tokyo 101
Tel: (03) 256-2211
TELEX: (03) 252-3774

JAPAN (Cont'd)

Ryoyo Electric Corporation
Shuwa Sakurabashi Bldg.
4-5-4 Hatchobori
Chuo-Ku, Tokyo 104
Tel: (03) 555-4811

Tokyo Electron Ltd.
Shinjuku Nomura Bldg.
1-26-2 Nishi-Shinjuku
Shinjuku-Ku, Tokyo 160
Tel: (03) 343-4411
TELEX: 232-2220 LABTEL J

KOREA

J-TEK Corporation
2nd Floor, Government Pension Bldg.
24-3 Yoido-Dong
Youngdungpo-Ku
Seoul 150
Tel: 011-82-2-782-8039
TELEX: KODIGIT K25299

Samsung
23rd Fl. Dong Bang Bldg.
1502-KA Taepyung-RU
Chung-Ku
Seoul
Tel: 777-78
TELEX 27970 KORSST K

MEXICO

DICOPEL S.A.
Tochtli 368 Fracc. Ind. San Antonio
Azcapotzalco
C.P. 02760-Mexico, D.F.
Tel: 90115255613211
TELEX: 1773790 DICOME

NEW ZEALAND

Northrup Instruments & Systems Ltd.
459 Kyber Pass Road
P.O. Box 9464, Newmarket
Auckland 1
Tel: 011-64-9-501-219, 501-801, 587-037
TELEX: NZ21570 THERMAL

Northrup Instruments & Systems Ltd.
P.O. Box 2406
Wellington 856658
TELEX: NZ3380

PAKISTAN

Computer Applications Ltd.
7D Gizri Boulevard
Defence
Karachi-46
Tel: 011-92-21-530-306
TELEX: 24434 GAFAR PK

Horizon Training Co., Inc. (Agent)
1 Lafayette Center
1120 20th Street N.W.
Suite 530
Washington, D.C. 20036
Tel: (202) 887-1900
TWX: 248890 HORN

SINGAPORE

General Engineers Corporation Pty. Ltd.
203 Henderson Road
1102 Henderson Industrial Park 0315
Tel: 011065-271-3163
TELEX: RS23987 GENERCO

SOUTH AFRICA

Electronic Building Elements, Pty. Ltd.
(Mailing Address)
P.O. Box 4609
Pretoria 0001
Tel: 011-27-12-469921
TELEX: 3-22786 SA

(Shipping Address)
Pine Square, 18th Street
Hazelwood Pretoria

TAIWAN

Mitac Corporation
No. 585 Ming Sheng E. Road
Taipei
Tel: 011-96-2-501-8231
TELEX: 11942 TAIAUTO

VENEZUELA

P. Benavides CA
Arilanes a Rio
Resdencias Kamarata
Local 4 a LZ
Caracas
Tel: (582) 571-0396

*Field Application Location

CG-11/6/86

intel

DOMESTIC SERVICE OFFICES

ALABAMA

Intel Corp.
5015 Bradford Drive, #2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Dr. #D214
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
500 E. Fry Blvd., Suite M-15
Sierra Vista 85635
Tel: (602) 459-5010

ARKANSAS

Intel Corp.
P.O. Box 206
Ulm 72170
Tel: (501) 241-3264

CALIFORNIA

Intel Corp.
21515 Vanowen
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: 1-800-468-3548

Intel Corp.
1900 Prairie City Rd.
Folsom 95630-9597
Tel: (916) 351-6143

Intel Corp.
1350 Shorebird Way
Mt. View 94043
Tel: (415) 968-8211
TWX: 910-339-9279
910-338-0255

Intel Corp.
2000 E. 4th Street
Suite 110
Santa Ana 92705
Tel: 1-800-468-3548
TWX: 910-595-2475

Intel Corp.
2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 970-1740

Intel Corp.
4350 Executive Drive
Suite 150
San Diego 92121
Tel: (619) 452-5880

COLORADO

Intel Corp.
650 South Cherry
Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
Danbury 06811
Tel: (203) 748-3130

FLORIDA

Intel Corp.
6363 N.W. 6th Way
Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

Intel Corp.
242 N. Westmonte Drive
Suite 105
Altamonte Springs 32714
Tel: (305) 869-5588

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 441-1171

ILLINOIS

Intel Corp.
300 N. Martingale Rd.
Suite 300
Schaumburg 60194
Tel: (312) 310-8034

INDIANA

Intel Corp.
8777 Purdue Rd., #125
Indianapolis 46268
Tel: (317) 875-0623

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 345-2727

KENTUCKY

Intel Corp.
3525 Tatescreek Road,
#51
Lexington 40502
Tel: (606) 272-6745

MARYLAND

Intel Corp.
4th Floor Product Service
7833 Walker Drive
Greenbelt 20770
Tel: (301) 220-3313

MASSACHUSETTS

Intel Corp.
3 Carlisle Road
Westford 01886
Tel: (617) 692-1060

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8905

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 143
Earth City 63045
Tel: (314) 291-2015

NEW JERSEY

Intel Corp.
385 Sylvan Avenue
Englewood Cliffs 07632
Tel: (201) 567-0820
TWX: 710-991-8593

Intel Corp.
Raritan Plaza III
Raritan Center
Edison 08817
Tel: (201) 225-3000

NORTH CAROLINA

Intel Corp.
2306 W. Meadowview Road
Suite 206
Greensboro 27407
Tel: (919) 294-1541

OHIO

Intel Corp.
Chagrin-Brainard Bldg.
Suite 305
28001 Chagrin Boulevard
Cleveland 44122
Tel: (216) 464-6915
TWX: 810-427-9298

Intel Corp.
6500 Poe
Dayton 45414
Tel: (513) 890-5350

OREGON

Intel Corp.
10700 S.W. Beaverton-Hillsdale
Highway
Suite 22
Beaverton 97005
Tel: (503) 641-8086
TWX: 910-467-8741

Intel Corp.
5200 N.E. Elam Young Parkway
Hillsboro 97123
Tel: (503) 681-8080

PENNSYLVANIA

Intel Corp.
201 Penn Center Boulevard
Suite 301 W
Pittsburgh 15235
Tel: (313) 354-1540

TEXAS

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628
TWX: 910-874-1347

Intel Corp.
12300 Ford Road
Suite 380
Dallas 75234
Tel: (214) 241-2820
TWX: 910-860-5617

Intel Corp.
8815 Dyer St., Suite 225
El Paso 79904
Tel: (915) 751-0186

VIRGINIA

Intel Corp.
1603 Santa Rosa Rd., #109
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
110 110th Avenue N.E.
Suite 510
Bellevue 98004
Tel: 1-800-468-3548
TWX: 910-443-3002

WISCONSIN

Intel Corp.
450 N. Sunnyslope Road
Suite 130
Brookfield 53005
Tel: (414) 784-8087

CANADA

Intel Corp.
190 Attwell Drive, Suite 103
Rexdale, Ontario
Canada M9W 6H8
Tel: (416) 675-2105

Intel Corp.
620 St. Jean Blvd.
Pointe Claire, Quebec
Canada H9R 3K2
Tel: (514) 694-9130

Intel Corp.
2650 Queensview Drive, #250
Ottawa, Ontario,
Canada K2B 8H6
Tel: (613) 829-9714

CUSTOMER TRAINING CENTERS

CALIFORNIA

2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 970-1700

ILLINOIS

300 N. Martingale, #300
Schaumburg 60173
Tel: (312) 310-5700

MASSACHUSETTS

3 Carlisle Road
Westford 01886
Tel: (617) 692-1000

MARYLAND

7833 Walker Dr., 4th Floor
Greenbelt 20770
Tel: (301) 220-3380

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5–6 Tokodai Toyosato-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

FRANCE
Intel Paris
1 Rue Edison, BP 303
78054 Saint-Quentin en Yvelines
France

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Piper's Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel Semiconductor GmbH
Seidlstrasse 27
D-8000 Munchen 2
West Germany

HONG KONG
Intel Semiconductor Ltd.
1701 Connaught Centre
1 Connaught Road
Hong Kong

Printed in U.S.A./0187/50K/RRD VC
Development Tools

ISBN 1-55512-063-6