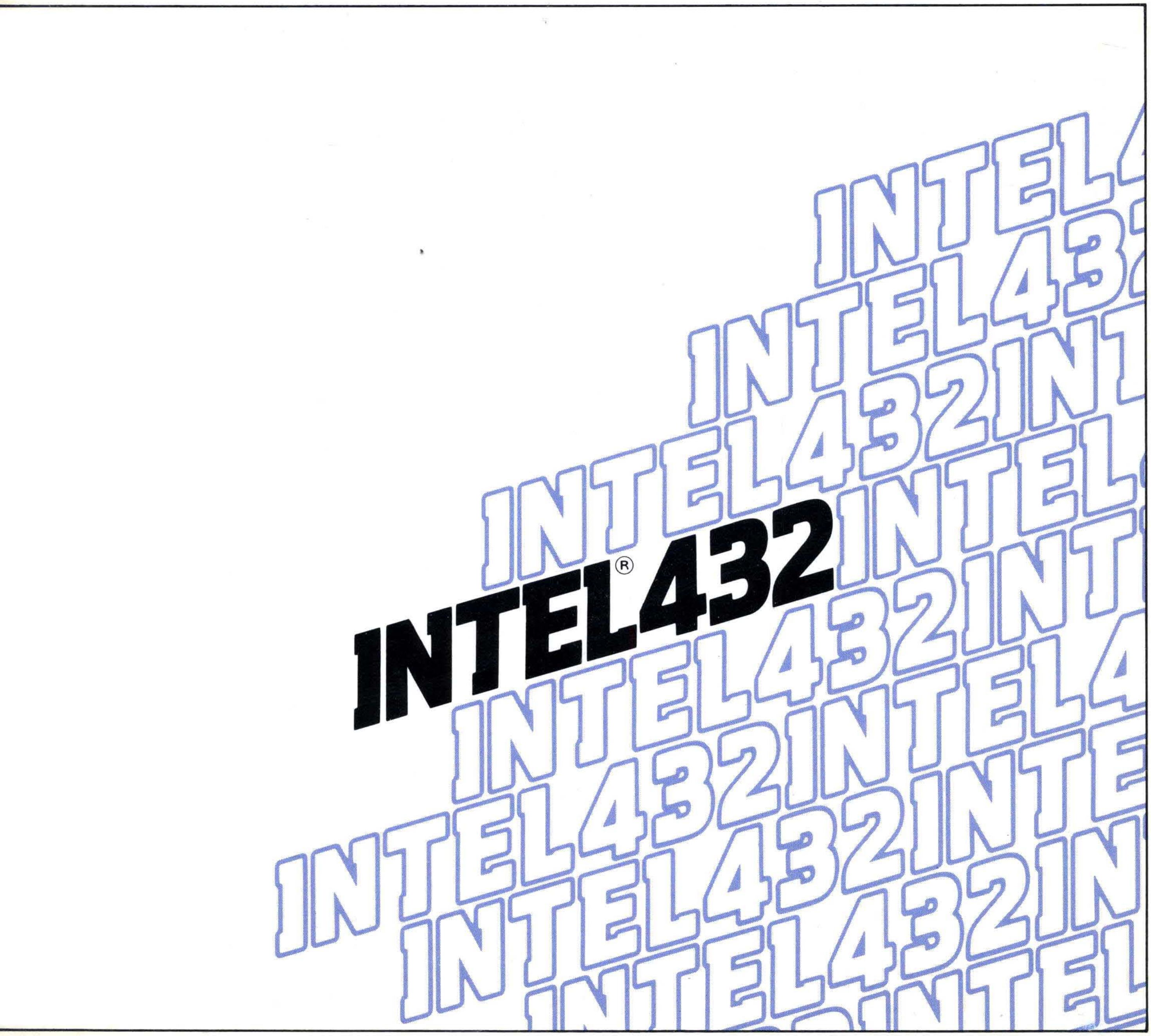
intel®

Intel 432 CDS Workstation User's Guide



INTEL 432
CROSS DEVELOPMENT SYSTEM
WORKSTATION USER'S GUIDE

Order Number: 172097-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

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

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP
CREDIT
i^{R}
ICE
iCS
iMMX
Insite
Intel^{R}
int$_{e}$l^{R}

Intelevision
Intellec^{R}
iOSP
iRMX
iSBC
iSBX
Library Manager
MCS^{R}
Megachassis

Micromainframe
Micromap
Multibus^{R}
Multimodule
Plug-A-Bubble
PROMPT
RMX/80
System 2000^{R}
UPI

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

Ada is a trademark of the Department of Defense (Ada Joint Program Office).

VAX and VMS are trademarks of Digital Equipment Corporation.

REV.	REVISION HISTORY	DATE
-001	Original issue	12/81

PREFACE

This document is the reference manual for DEBUG-432 and UPDATE-432, two software utilities in the Intel 432 Cross Development System for the Intel 432 Micromainframe computer system. Both of these utilities reside on the Intellec Series III Microprocessor Development System. The other principal utilities required for 432 program development -- the Ada* Compiler System and LINK-432 -- reside on the mainframe host.

This manual is directed to all users and potential users of DEBUG-432 and UPDATE-432. All references to the System 432/670 in this manual also apply to the System 432/671, a functionally identical unit that operates from a 230V/50Hz AC line. The System 432/670 operates from a 120V/60Hz AC line.

ORGANIZATION OF THIS MANUAL

DEBUG-432 and UPDATE-432 are relatively independent programs. Consequently, this document is divided into two parts. Part I discusses DEBUG-432; Part II discusses UPDATE-432.

The discussion is divided into nine chapters and eight appendixes:

Chapter 1 Overview of the Series III System

Gives an overview of debug workstation, whose hardware includes the Intellec Series III system and the System 432/670, and whose software includes DEBUG-432 and UPDATE-432.

PART I: DEBUG-432

Chapter 2 Overview of DEBUG-432

Introduces the DEBUG-432 program.

Chapter 3 Getting Started Using DEBUG-432

Discusses initialization issues, such as invoking the debugger and initializing the System 432/670. Also introduces the debugger environment and how to prepare for debugging programs.

Chapter 4 Examining and Modifying Memory Locations

Discusses using the debugger to examine and modify memory, including the use of symbolic names and presupplied templates.

*Ada is a trademark of the Department of Defense (Ada Joint Program Office).

Chapter 5 Processes, Contexts, and Breakpoints

Discusses breakpoints and the dynamic features of the debugger: selecting the current process, setting and removing breakpoints, and examining the call stack.

Chapter 6 Defining Templates

Shows how to define debugger templates and use them to examine and modify objects in memory.

Chapter 7 DEBUG-432 Commands

Gives complete syntax for all DEBUG-432 commands. The commands are arranged alphabetically for easy reference.

Part II: UPDATE-432

Chapter 8 Overview of UPDATE-432

Introduces the UPDATE-432 program.

Chapter 9 Using UPDATE-432

Shows how to invoke UPDATE-432, and discusses the parameters supplied to the program.

Appendices A through F contain miscellaneous useful information about DEBUG-432, including: a formal definition of the DEBUG-432 command syntax, a short description of all commands, the default template definitions, a log file of a sample debugging session, a discussion of how to debug faults, and the error messages. Appendix F gives a detailed description of possible error messages and their causes.

Appendices G and H contain UPDATE-432 syntax description and error messages.

RELATED PUBLICATIONS

In addition to this document, two other documents directly describe the use of the Intel 432 Cross Development System:

Introduction to the Intel 432 Cross Development System, Order Number: 171954. Introduces the 432 Cross Development System software and hardware. This manual contains a start-to-finish example of creating and debugging a 432 program.

Intel 432 Cross Development System VAX/VMS Host User's Guide, Order Number: 171870. Shows how to create, compile, and link programs at the VAX host, and describes downloading linked EODs to the debug workstation.

For more information regarding the iAPX 432 architecture, see the following manuals:

Introduction to the iAPX 432 Architecture, Order Number: 171821. Provides a comprehensive overview of the 432 architecture.

iAPX 432 Object Primer, Order Number: 171858. Introduces the major features of the 432 architecture.

iAPX 432 General Data Processor Architecture Reference Manual, Order Number: 171860. Contains complete, detailed descriptions of all aspects of the 432 General Data Processor architecture.

iAPX 432 Interface Processor Architecture Reference Manual, Order Number: 171863. Contains complete, detailed descriptions of all aspects of the 432 Interface Processor architecture.

The following reference manuals describe the Intellec Series III Microcomputer Development System, which serves as the intelligent console interface to the 432 debug workstation:

Intellec Series III Microcomputer Development System Console Operating Instructions, Order Number: 121609. Describes the operation and support utilities of the Intellec Series III system.

Intellec Series III Microcomputer Development System Programmer's Reference Manual, Order Number: 121618. Provides background and reference material concerning the use of the Intellec Series III system.

The System 432/670 and 671 are described in the following manuals:

System 432/600 System Reference Manual, Order Number: 172098. Defines the physical and functional characteristics of the System 432/600.

System 432/600 Diagnostic Software User's Guide, Order Number: 172099. Describes the capabilities and uses of the diagnostic software for the System 432/600.

The Ada language, the 432 extensions to Ada, and the iMAX 432 operating system are described in the documents listed below.

This compiler is presently an incomplete implementation of the Ada programming language. It is intended that this compiler will be further developed to enable implementation of the complete Ada programming language, and then be submitted to the Ada Joint Program Office for validation.

Reference Manual for the Ada Programming Language, Order Number: 171869. Defines the Ada programming language.

Reference Manual for the Intel 432 Extensions to Ada, Order Number: 172283. Defines Intel's extensions to the Ada programming language.

iMAX 432 Reference Manual, Order Number: 172103. Describes the features and functions of the iMAX 432 operating system.

Finally, the communication links between the VAX system and the Intellec Series III Microcomputer Development System are described in these publications:

Mainframe Link for Distributed Development User's Guide, Order Number: 121565. Describes the use of the synchronous communication link on the Intellec Series III system.

Asynchronous Communication Link User's Guide, Order Number: 172174. Describes the use of the asynchronous communication link.

NOTATIONAL CONVENTIONS

The following conventions are used in displaying command and control syntax in this manual:

CAPITALS Information in capitals must be entered as shown in the syntax statements. Although this information is shown in upper case, it may be entered in upper case or lower case.

variables Information in underscored lower case represents variable information that must be supplied when entering commands.

[] Brackets indicate parameters or controls that are optional.

{} Braces indicate a choice. Exactly one of the items enclosed in braces must be chosen.

¦ The vertical bar indicates a choice. It is usually used within braces to separate choices.

... The ellipsis indicates that multiple items may be entered.

<RETURN> Indicates the RETURN key.

CHAPTER 1 Page
OVERVIEW OF THE 432 SERIES III SOFTWARE

PART I -- DEBUG-432

CHAPTER 2
OVERVIEW OF DEBUG-432

CHAPTER 3
GETTING STARTED WITH DEBUG-432

CHAPTER 4
EXAMINING AND MODIFYING MEMORY

CHAPTER 5 Page
PROCESSES, CONTEXTS, AND BREAKPOINTS

CHAPTER 6
TEMPLATES

CHAPTER 7
DEBUG-432 COMMANDS

CHAPTER 9 Page
USING UPDATE-432

APPENDIX A
FORMAL DEFINITION OF DEBUG-432 COMMAND SYNTAX

APPENDIX B
DEBUG-432 RESERVED WORDS AND COMMAND SUMMARY

APPENDIX C
DEBUG-432 TEMPLATES IN DEB432.TEM

APPENDIX D
SAMPLE DEBUG-432 SESSIONS

APPENDIX E Page
HOW TO DEBUG FAULTS

APPENDIX F
DEBUG-432 ERROR MESSAGES AND PROBABLE CAUSES

APPENDIX G
FORMAL DEFINITION OF UPDATE-432 COMMAND SYNTAX

APPENDIX H
UPDATE-432 ERROR MESSAGES

ILLUSTRATIONS

TABLES

CHAPTER 1
OVERVIEW OF THE 432 SERIES III SOFTWARE

INTRODUCTION

The Ada program development environment for Intel 432 Micromainframe computer systems is the Intel 432 Cross Development System (432 CDS). This development environment consists of three major components:

- A VAX mainframe host computer on which Ada programs are edited, compiled, and linked
- An Intellec Series III Microcomputer Development System
- A System 432/670 execution vehicle

The Series III and System 432/670 together compose the debug workstation described in this manual. Linked Ada programs are downloaded from the host to the workstation, where they can be debugged and updated.

The 432 Cross Development System software includes programs that reside on the mainframe host and programs that reside at the workstation on the Series III system. The host software includes the Ada Compiler System, the LINK-432 linker, and the iMAX operating system. The Series III 432 software, described in this manual, includes the DEBUG-432 debugger and the UPDATE-432 linked program updater. Users of 432 Series III software may also use all standard ISIS-II utilities.

Figure 1-1 shows a schematic representation of the Intel 432 Cross Development System environment. The diagram indicates which software resides on each of the three systems and the overall organization of the program development process.

After users have compiled their Ada programs and linked the resultant EODs, they download the EODs to the debug workstation. Either the synchronous communications link or the asynchronous communications link is used for downloading; see the communications link manuals for instructions.

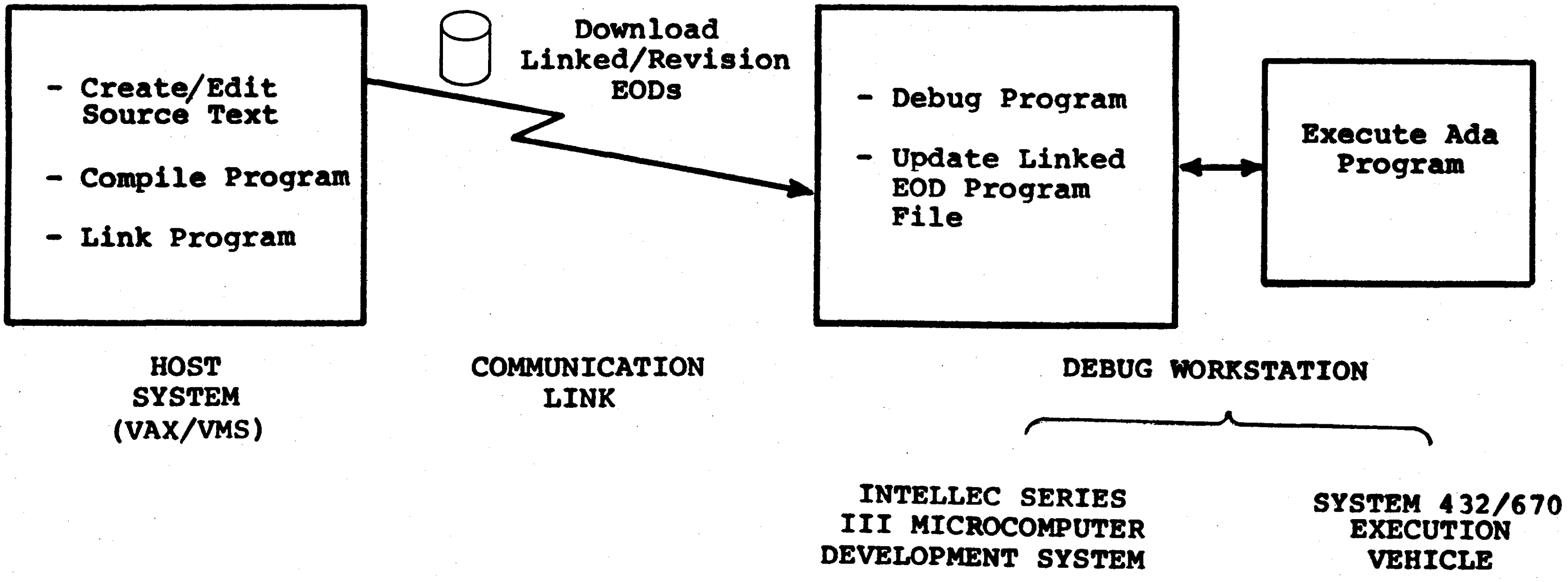

- Create/Edit Source Text
- Compile Program
- Link Program
Download Linked/Revision EODs
- Debug Program
- Update Linked EOD Program File
Execute Ada Program
HOST SYSTEM (VAX/VMS)
COMMUNICATION LINK
DEBUG WORKSTATION
INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM
SYSTEM 432/670 EXECUTION VEHICLE
F-0043

Figure 1-1. The Intel 432 Cross Development System

THE INTELLEC SERIES III SYSTEM

The Intellec Series III Microcomputer Development System, one component of the 432 CDS, contains a minimum of 192K bytes of RAM, a disk storage system (at least one 8M-byte cartridge disk drive), a console interface, and an iSBC 432/602 Interface Processor board which is part of the Intellec Series III/432 Interconnect Kit. The Interconnect Kit provides the logical and electrical interconnection between the Series III system and the System 432/670.

The Series III runs the ISIS-II operating system, which handles file management, and the RUN 8086 program, which is used to execute 8086 programs.

Both DEBUG-432 and UPDATE-432 are 8086 programs, executed using RUN.

For more information on the Series III system, see the Intellec Series III Microcomputer Development System Console Operating Instructions manual.

DEBUG-432

The iAPX 432 system debugger, DEBUG-432, provides an environment in which users can test their 432 Ada programs and identify errors for subsequent correction. The debugger lets the user initialize the System 432/670, load a linked EOD into the System 432/670 for execution, and test the program in a controlled environment. It provides mechanisms for interactively starting, halting, and monitoring the program being executed. The user monitors the program by halting execution at preset program locations called breakpoints, and then examining and modifying the current state of System 432/670 main memory. Portions of System 432/670 memory can also be saved in ISIS-II files. Finally, using DEBUG-432 the user can examine the memory state of a crashed System 432/670 execution vehicle.

Part I of this manual describes DEBUG-432. Also see Introduction to the Intel 432 Cross Development System for an example of program development and debugger use.

UPDATE-432

UPDATE-432 lets users avoid some downloading time when sending revised Ada programs from the host to the workstation. When an Ada module is modified and the compiled EOD is relinked, it is not necessary to transmit the entire newly linked EOD over the communications link; instead only a file describing the changed modules (a revision EOD) need be sent. Since the revision EOD contains only differences between the newly linked EOD and the previously linked EOD, it can be significantly shorter than a linked EOD. UPDATE-432 modifies the current linked EOD file on the Series III based on information in the revision EOD.

Updating revises the program modules and the segment addresses. It can add or delete modules to a linked EOD. The names of the modules to be deleted, and the actual modules to be inserted, are found in the revision EOD.

Part II of this manual describes UPDATE-432.

RUN-TIME ENVIRONMENT OF WORKSTATION SOFTWARE

HARDWARE ENVIRONMENT

The workstation software requires the following hardware configuration:

- Intellec Series III Microcomputer Development System including
  - At least 192K bytes of memory
  - At least one hard disk drive
  - A system console device
- An Intellec Series III/432 Interconnect Kit to connect the Series III and System 432/670
- A System 432/670 execution vehicle

The Intellec Series III system can be a factory-delivered Series III or a Series II upgraded to a Series III

SOFTWARE ENVIRONMENT

The user also needs the following software configuration:

- ISIS-II/86, the Series III operating system, including the RUN program, V1.1 or later
- DEB432.86, the executable debugger
- DEB432.TEM, an Intel-supplied set of templates
- UPDATE.86, the executable updater

ISIS-II/86 is part of the Series III software package. The other software is supplied on the Intel 432 Cross Development System (Intel 432 CDS) software distribution diskette.

This manual assumes installation of the Intel 432 CDS was done according to the instructions in the Introduction to the Intel 432 CDS. Thus, the Intel-supplied files are assumed to be on diskette drive :f0.

A small portion of the debugger resides in the System 432/670 main memory (Throughout this manual, the term "432 memory" will refer to the main memory on the System Bus of the System 432/670 execution vehicle). This part of the debugger is included in the iMAX 432 operating system, which is linked in with other Ada object modules by the Intel 432 CDS linker, LINK-432. See the Introduction to the Intel 432 CDS and the Intel 432 CDS VAX/VMS Host User's Guide for more details.

PART I

DEBUG-432

CHAPTER 2
OVERVIEW OF DEBUG-432

INTRODUCTION

Debugging is the process of detecting, locating, and removing mistakes from a program. To debug an Ada program, a 432 CDS user executes that program on the System 432/670 under control of the system debugger, DEBUG-432. The programmer can then interactively monitor the program to see how it behaves; he can halt execution at preset program locations (breakpoints), and can examine and modify the state of System 432/670 main memory (hereafter 432 memory) at any time.

Debugging consists of several steps:

1. Using DEBUG-432, the user initializes the System 432/670 and loads 432 memory.

2. Before starting the System 432/670, the user may examine and perhaps modify 432 memory. The user can set breakpoints in instruction segments. Finally, the user starts the system.

3. The user may examine and modify 432 memory while the processes being debugged are running, or may wait until one or more processes reach a breakpoint.

   Additional breakpoints can be set, existing ones removed, and processes at breakpoints can be resumed. By setting breakpoints and examining 432 memory, the user finds some of the bugs in his program.

4. When changes are required, the source program is modified and the program is recompiled and relinked. Using the communication link, a revision EOD is downloaded to the Series III system. UPDATE-432 is used to update the linked EOD. Then the user returns to step 1.

5. Some changes can be tested before revising the source code. One way to test changes is by patching (modifying) the 432 memory image immediately after loading it, before starting the system. Another way is to wait until a process hits a particular breakpoint and then patch memory.

This document provides instructions on carrying out all these steps. The debugger also provides special mechanisms to deal with those special features of the iAPX 432 architecture not mentioned in this simple procedure, in particular, the organization of data into objects and the use of multiple processes and processors.

DEBUG-432 FEATURES

DEBUG-432 provides the conventional features usually associated with systems-level debuggers, along with support for objects and multiple processes, which are features of the unique iAPX 432 environment.

CONVENTIONAL FEATURES

Among the conventional features offered by DEBUG-432 are the following:

- examining and modifying memory locations
- setting and removing breakpoints
- saving and restoring memory images
- logging a debugger session

SUPPORT FOR OBJECTS

The iAPX 432 has an object-oriented architecture, which means that processors can directly manipulate hardware-protected data structures called objects. The addressing method supported by DEBUG-432, along with a template mechanism, lets users examine and modify objects as well as absolute memory locations. Provided with the debugger are several standard templates that specify some system objects; users can change these templates and define additional ones for their own objects.

SUPPORT FOR MULTIPLE PROCESSES

The iAPX 432 architecture directly supports multi-process systems. DEBUG-432 lets the user interact with multiple processes as well as the procedures in these processes. The objects associated with a process can be examined while the process is executing; it is not necessary to breakpoint the process. Users can set breakpoints for one, several, or all processes in the system.

The debugger lets the user easily focus on one process while debugging in a multiprocess environment.

CONCURRENT DEBUGGING AND PROGRAM EXECUTION

DEBUG-432 and the user's 432 program run asynchronously, using completely separate processors. Thus the debugger and user program execute independently. The user can examine and modify memory while the processes being debugged are running. Even if the user's 432 program begins to loop, the debugger is not affected and can continue to debug the program.

CHAPTER 3
GETTING STARTED WITH DEBUG-432

This chapter shows how to invoke the debugger from an ISIS-II/86 system. It then discusses the format of DEBUG-432 commands, simple command editing, special control characters, and DEBUG-432 I/O modes.

INVOKING DEBUG-432

DEBUG-432 executes under control of the RUN 8086 program. To invoke DEBUG-432, first invoke RUN.

The Series III has two execution modes: the 8080/8085 mode under control of ISIS-II, and the 8086 mode under control of RUN 8086. When the Series III system starts running, it is in 8080/8085 mode.
To invoke the debugger from 8080/8085 mode, type:

```
    -RUN [:Fn:]DEB432
```

where :Fn: refers to the disk drive which contains the debugger in file DEB432.86.

This loads RUN, activates 8086 mode, loads DEB432.86, and starts execution of the debugger.

RUN maintains a default workfile drive number; if the default drive is not available, the debugger displays its sign-on message, but then the RUN program fails and displays:

```
    ISIS ERROR 30 USER PC XXXX

    FATAL ISIS ERROR - RUN TERMINATED
```

The ISIS-II system reboots and returns to 8080/8085 mode. At this point, make sure that the setting of the default workfile drive is correct for your hardware configuration. In other words, verify that the default drive specified for workfiles in your system is available, has space, and is not write protected. To find the current default drive setting, type:

```
    RUN WORK
```

To set the drive number, type:

```
    RUN WORK :fn:
```

where n is the drive number (from 0 to 9).

The debugger's log file uses the current setting of DATE obtained from the RUN program. To set the date, type:

```
    RUN DATE mm/dd/yy
```

where mm is the month, dd the day, and yy the year.

For more information on the RUN program, and the DATE and WORK commands, see the Intellec Series III Microcomputer Development System Console Operating Instructions.

After you invoke it, DEBUG-432 signs on with the message:

```
    SERIES III  432 SYSTEMS LEVEL DEBUGGER, Vx.yz

    ?
```

The ? is the debugger's prompt, indicating the debugger is ready to accept commands from the console. The current revision number of the debugger is indicated by x.yz.

DEBUGGER COMMAND SYNTAX

This section describes the general format of debugger command lines, comments, and command syntax errors.

The general format of a DEBUG-432 command line is:

```
    command[;[command]]... <RETURN>
```

You can enter debugger commands on a line to a maximum of 127 characters per line. The RETURN key indicates the end of the command line and instructs the debugger to execute the commands in the order in which they appear.

To continue a command over more than one line, type an ampersand (&) before the RETURN. DEBUG-432 treats the ampersand-RETURN pair as a blank.

The debugger permits comments in a command line. Comments are signaled by the presence of a double hyphen (--), which terminates the command, or an ampersand (&), which tells the debugger to continue this command line on the next physical line. The prompt for the second and following lines of a command is a double question mark (??). DEBUG-432 ignores any characters that follow the double hyphen or ampersand.

For example:

```
    ?command          --This is an example of a comment
    ?command1&          This comment is in the middle of a command
    ??parameter1
    ?
```

The debugger is case insensitive; it does not distinguish between capital and lower case letters. Thus

```
    ?COMMAND
```

and

```
    ?command
```

have the same meaning to the debugger.

Command Syntax Errors:

The debugger parses and executes each command in the line after the RETURN key has been typed. If a syntax error occurs -- that is, if the debugger encounters a command that it does not understand -- it reports an error on the line following the command and ignores subsequent commands. The debugger also positions a caret under the character in the command line where it discovered the syntax error. For example:

```
    ?(1 + 2 - 3
    (1 + 2 - 3
              ^
    ERR 232: ILLEGAL SYNTAX
    ?
```

In the above example, the error cursor is placed at the end of the command line because the RETURN at the end of the line should not be present within parentheses.

SPECIAL CHARACTERS

All non-printing characters (except for a few special cases, discussed in the next section) are echoed to the console as question marks. For example, if you type a CONTROL-Y, the debugger displays a question mark indicating that a non-printing character has been entered. Some characters (CONTROL-B, CONTROL-O) are significant to the debugger if they are at the beginning of a command line but are echoed as question marks if they appear anywhere else in the line.

Several non-printing characters are useful for editing command lines, interrupting the debugger, changing I/O modes, and controlling the screen display. The following sections describe these special characters in detail.

COMMAND EDITING

Line editing lets the user correct typing errors in the command line. Each character typed is stored in an editing buffer until the user presses the RETURN key. While the characters are stored in the editing buffer, they may be edited through special non-printing editing characters:

RUBOUT erases the last character typed from both the screen and edit buffer.

CONTROL-X erases the entire line currently being input; that is, erases up to the last RETURN.

CONTROL-R retypes the current line being typed. If the current line is empty, echoes the previous command line and positions the cursor at the end of the line. You can edit the line by using RUBOUT, or execute it by typing RETURN.

CONTROL-C aborts the entire command line (also changes the debugger I/O mode as described later in this chapter)

CONTROLLING CONSOLE OUTPUT

The ISIS-II operating system lets you suspend and resume console output (without losing any output in the process) with the following control characters:

CONTROL-S suspends console output

CONTROL-Q resumes console output

CONTROL-D

The RUN program, under which DEBUG-432 executes, uses CONTROL-D to start the 8086 debugger on the currently running program (in this case, DEBUG-432). If you press CONTROL-D, the following message appears:

```
PROCESSING ABORTED
*
```

where "PROCESSING ABORTED" is a message from the 8086 debugger and the asterisk prompt (*) indicates that the 8086 debugger is ready to accept commands. Typing GO<RETURN> returns you to the 432 debugger (you may have to type a RETURN to get a prompt from the 432 debugger at this point). Typing EXIT would leave the 8086 debugger but would also abort your DEBUG-432 session, returning you to ISIS-II.

CHANGING DEBUGGER I/O MODES AND INTERRUPTING DEBUG-432

To interrupt any command and return control to the debugger, enter a CONTROL-C. The debugger displays the following message and prompts for a new command:

```
?DEBUGGING ONLY
?
```

The message "DEBUGGING ONLY" indicates that the debugger is in debugging only mode. This is one of three debugger I/O modes discussed in the next section. The other characters that control I/O modes are CONTROL-O and CONTROL-B:

CONTROL-C	place the debugger in Debugging Only mode
CONTROL-O	place the debugger in I/O Only mode
CONTROL-B	place the debugger in Debugging + I/O mode

CONTROL-O and CONTROL-B perform their functions only if they are the first character of a line, otherwise they echo as question marks. CONTROL-C, however, works under all circumstances; it interrupts debugger execution and begins a mode in which the user's 432 program cannot send output to the console.

Changing to a new mode displays an identifying message on the console. Also, unless in I/O Only mode, you can give the debugger's MODE command at any time to display the current mode of the debugger. For example, after a CONTROL-B the MODE command would produce the following:

```
?MODE
DEBUGGING + I/O
?
```

THE DEBUGGER I/O INTERFACE

The debugger provides a certain degree of support for I/O between Ada programs and the user. This section describes the I/O interface and the three I/O modes available.

For purposes of I/O, the debugger supports three interactive modes:

1. Debugging + I/O
2. Debugging Only
3. I/O Only

Debugging + I/O mode is the initial mode of the debugger.

The user may switch modes using the control characters described in the previous section.

DEBUGGING ONLY MODE

When the debugger is in Debugging Only mode, all debugging commands are available to the user, but no input to or output from the user's 432 program takes place. If the 432 program requests input or output, the request is not acknowledged until the user changes to another mode.

CAUTION

Immediately after 432 program execution is started, iMAX signs on using debugger-supported I/O. If the debugger is in Debugging Only mode, the operating system cannot sign on until the user changes modes.

Similarly, no 432 program can perform I/O if the debugger is in Debugging Only mode.

DEBUGGING + I/O MODE

Debugging + I/O mode supports program I/O in addition to debugger commands. In Debugging + I/O mode, data written from the executing 432 program is displayed on the console, and input data may be sent to the program by the user.

A write request from the 432 program is satisfied when the user is "at a prompt." The user arrives at a prompt by executing a command or by keying in either a CONTROL-X or enough RUBOUTs to get back to the start of a command line. If necessary, write requests from the System 432/670 are held until the user is at a prompt.

In Debugging + I/O mode, date written to the console from the executing 432 program is preceded by the identifying message

```
-- FROM 432:
```

Although no prompt is issued after the write operation, the user is free to enter debugger commands. Subsequent output from the 432 program will not have the identifying message preceding it until the user enters a debugger command.

To direct input to the 432 program, type a percent sign (%) as the first character of the input line. Ordinary line edit keys, such as RUBOUT and CONTROL-X, may be used when entering such an input line. When the RETURN key is pressed, the characters to the right of the percent sign are sent to the 432 program, along with a trailing <CR><LF> pair (obtained by pressing RETURN).

I/O ONLY MODE

In this mode, the user is interacting with the 432 program only and cannot give debugger commands; all input is assumed to be directed to the 432 process. No debugger prompts are displayed. Furthermore, input lines to the 432 program should not be preceded with "%", nor is output from the program preceded with the "--FROM 432:" message.

Users running complete, correct 432 programs should choose this mode.

ENTERING INPUT LINES

The debugger I/O interface defines a 132-character buffer for input to the 432 program. Each time the user keys in a line as input for the 432 (either in I/O Only or in Debugger + I/O mode), the line is placed in this buffer. If there were previously characters in the buffer, the current line is placed after those characters. This permits type-ahead of up to 132 characters.

Whenever the 432 program makes a read request for n characters, it is satisfied by the characters remaining in the first line of the buffer. If the line does not have n characters, whatever characters remain are used to satisfy the request; the next read request uses the next line in the buffer. If the buffer is empty when a read request is given, the request is not satisfied until the user keys in a line of input to the 432 program.

The minimum number of characters that can be placed into the buffer is two: namely, a <CR><LF> pair obtained by typing a RETURN. The 432 program may issue a read request for a maximum of 132 characters.

When the I/O buffer is full, no further characters are accepted from the user. If a character is entered when the buffer is full, the debugger sounds the console bell (CONTROL-G); in this case, the input character is not echoed.

When the debugger is in Debugging + I/O mode and is accepting commands from an include file (see the INCLUDE command), it does not support type-ahead of 432 input lines. After each line of 432 program input (preceded by %) is read, the debugger stops reading from the include file until the 432 program has read the entire line. Although the debugger stops reading from the include file, it is still polling for breakpoints and 432 I/O requests. To interrupt an include file and return control to the console, type CONTROL-C.

BREAKPOINT ANNOUNCEMENTS AND DEBUGGER MODES

Breakpoints are always announced, regardless of the debugger mode. If the user is at a prompt, the a breakpoint is announced as described in Chapter 5. On the other hand, if the user is entering a command or input line when a breakpoint occurs, the console bell is sounded; subsequently, when the user reaches a prompt, the breakpoint message appears.

PREPARING THE DEBUGGING ENVIRONMENT

The debugger I/O mode is one of several parameters that determine the debugging environment -- the context within which a user interacts with the debugger. This debugging environment affects what commands the user can give at any time, what the commands will do, and how the results are displayed. Other parameters in the environment are the:

- default numeric input base
- default numeric output base
- log file (if any)
- log state (logging only, console output only, or both console output and logging)
- top of System 432/670 memory

The level of 432 memory addressing available at any time is also part of the environment. In various states the debugger supports logical and physical addressing, physical addressing only, or no addressing at all.

The user can maintain a set of symbolic names in the debugger name table; this set of names is also considered part of the debugging environment.

The rest of this chapter explains the various parts of the debugging environment and how to control them. Setting up a workable environment is a prerequisite to debugging a program.

The INCLUDE command can read in a file that contains a set of debugger commands to establish the debugging environment.

GLOBAL DEBUGGING PARAMETERS

The following sections describe the parameters listed above and are summarized in Table 3-1 below.

Default Input and Output Bases

The debugger does 32-bit, twos-complement arithmetic. Integers are similar in format to integers and based numbers in Ada. The difference between Ada format and debugger format is that the default base in Ada is 10, whereas the default base in the debugger may be changed.

When the debugger signs on, the default input and output bases are base 10. That is, all numbers which the user types in as part of a command, or which the debugger prints out in response to a command, are in base 10. Base 10 is typically used for defining templates (see Chapter 6). Actual debugging is usually done in base 16; the file DEB432.TEM, which the user INCLUDEs to define the standard templates, contains commands to set the input and output bases to 16.

At any time, a based number, written in the form: base#number# overrides the default base. The base will always be a decimal number. For example, in base 10, typing 16#FE# is the same as typing 254.

If the input base is changed to 16, typing 10#254# is the same as typing OFE.

When a number begins with a letter and is not in based number format, it must be preceded with 0 to distinguish it from an identifier.

The default input base is changed using

```
SUFFIX n
```

The default output base is changed with

```
BASE n
```

where n is the desired base, either in decimal or in the form base#number#.

Type BASE to find the current default output base, and SUFFIX to find the current default input base.

The I/O Mode

The three debugger I/O modes were discussed above under "The Debugger I/O Interface". The I/O mode determines the way the debugger interprets user input: as debugger commands or as 432 input. It also allows the debugger to "disappear from view": in I/O only mode the debugger transmits all input to the System 432/670. In Debugging Only mode, all input goes to the debugger. Debugging Only mode might be used to suspend the output of other, executing, processes while a breakpointed process is being debugged.

The Log File

The logging facility can record all or part of a debugging session. The command

```
LOG filename
```

opens a log file and turns logging on. Any previous log file is closed and the new one opened. While logging is on, all debugger input and output that appears at the console is copied into the log file. This includes I/O to and from the System 432/670.

After a log file has been opened, the user can give commands to send output to the console only, the log file only, or both:

```
>CRT                 -- direct output to the console only

>LOG                 -- direct output to the log file only

LOG                  -- resume suspended logging: direct output to
                     -- both log file and console
```

Once a log file has been opened in a debugging session, a log file will be open for the remainder of the session. Type VERSION to find the current log file name and whether logging is ON or OFF.

Top of Memory

The top of memory is the address of the last byte in the system. The debugger needs to know the top of System 432/670 memory to check the validity of addresses used later in the debugging session. It also uses the top of memory to locate the 256-byte window that controls the System 432/670 Interface Processor.

After the System 432/670 is powered on, the INIT command must be used to reset the System 432/670 and clear memory (see "The Debugger Addressing State," below). The debugger determines the address of the last byte of 432 memory and displays it as part of the INIT command:

```
    ?INIT
    TOP OF MEMORY: 7FFFF
```

If INIT SYSTEM is used, the debugger does not clear nor determine the top of memory. Instead, the user supplies the top of memory:

```
    INIT SYSTEM !number
```

where number is the address of the last byte of System 432/670 memory.

The VERSION command displays the current top of memory.

Table 3-1. Global Debugging Parameters

Parameter	Command to Set Parameter	Command to Display Current Setting
Default input base	SUFFIX n	SUFFIX
Default output base	BASE n	BASE
I/O mode		MODE, VERSION
Debugging Only	CONTROL-C	
Debugging + I/O	CONTROL-B	
I/O Only	CONTROL-O	
Log file	LOG filename	VERSION
Log state		VERSION
CRT and log file	LOG [filename]	
CRT only	>CRT	
Log file only	>LOG	
Top of memory	INIT INIT SYSTEM !number	VERSION

THE DEBUGGER ADDRESSING STATE

The addressing state refers to the kind of addressing the debugger will currently support: logical and physical, physical only, or none.

No Addressing

If 432 memory has not been initialized with an INIT command since the last power-up, or if memory is initialized but the debugger does not have the current top of memory value, the debugger cannot access memory and hence no addressing is permitted.

Use the INIT command to initialize memory if necessary. If the command is successful, physical addressing is allowed.

If memory has been initialized since the last power-up, but the debugger needs a top of memory value, use INIT SYS. The command resets the GDP, tells the debugger the location of the top of memory, and enables physical addressing without harming the existing image in memory.

Physical Addressing Allowed

Whenever the 432 memory has been initialized and the debugger knows the top of memory, physical addressing is allowed. That is, the user can reference memory by byte address. The user can also address the 432 interconnect space (see Chapter 4). Logical addresses may not be used. Neither the IPC or START commands may be used, nor breakpoints set, until logical addressing is enabled.

To allow logical addressing, in which memory is referenced by access descriptor coordinates (see Chapter 4), a load image must be provided with the LOAD, DEBUG, or RESTORE command.

The LOAD or RESTORE command must be followed by a DEBUG command. If the DEBUG command finds a consistent 432 object table directory, logical addressing is allowed.

A bad electrical connection between the Series III and System 432/670 could return the system to the no addressing state; use INIT or INIT SYSTEM as described above.

Logical Addressing Allowed

In this state, both physical and logical addresses may be used. All debugger commands are recognized, and the debugger polls for breakpoints and I/O.

If the INIT or INIT SYSTEM command is used in this state, or if a new memory image is loaded with the LOAD command, the debugger returns to physical addressing state.

A bad electrical connection between the Series III and the System 432/670 could return the system to the no addressing state; follow the steps outlined above to enable physical and logical addressing again.

NAMES

DEBUG-432 lets the user assign symbolic names to integers, memory references, breakpoints, and templates. These names are stored in a debugger-maintained structure called the Debugger Name Table (DNT).

Debugger names are of the same form as Ada names: an intial letter followed by letters, digits, and isolated underscores. The maximum length of a name is 128 characters.

A name may not have more than one definition in the DNT. If the user tries to define an existing name, the debugger issues an error. Reserved words may not be declared as names. A list of reserved words appears in Appendix B.

Declaring Names

Naming an integer permits use of the name in expressions; the user can enter the name instead of the integer itself. The syntax for declaring an integer name is:

```
    name: INTEGER [:= expression]
```

The expression is the value of the integer.

The name may then be used in expressions. For example:

```
    ?index:INTEGER := 4
    ?offset:INTEGER := index*16
```

References, breakpoints, and templates are named when they are defined. Chapter 4 describes references and naming a reference. Chapter 5 discusses breakpoints. Defining a breakpoint sets the breakpoint in 432 memory and stores its definition in the name table. Chapter 6 describes templates, used to display objects in memory. Having the template name in the name table makes that template available for use in displaying and modifying memory.

Listing Names in the DNT

The DIR command lists the names and definitions in the DNT alphabetically. The entire DNT can be listed, with a brief definition of each item, by typing:

```
    DIR
```

To list all the definitions of a particular type, use the DIR command followed by the appropriate type: BREAK, INTEGER, REFERENCE, or TEMPLATE. For example:

```
    DIR TEMPLATE
```

lists, in alphabetical order, all the template names, and a brief form of the definition of each (as much as will fit on one line).

To print just one item, type DIR followed by item name. This prints the complete definition of any name in the DNT.

Removing Names From the DNT

Any name in the DNT may be removed using the REMOVE command. Use:

```
    REMOVE ALL
```

to empty the entire directory.

To remove all names of a type, follow REMOVE by the type keyword. For example:

```
    REMOVE BREAK
```

removes all breakpoints from the DNT. Removing a breakpoint from the DNT also deactivates it in 432 memory.

To remove a single name, follow REMOVE with the name of the item to be removed. To redefine a name, REMOVE it and then reenter it.

Symbol Table Space

The debugger name table is stored in Series III memory and overflows to a disk workfile. To speed access to user symbols which have been paged out to disk, a hard disk drive should be used for the workfile. Fastest access to user symbols is obtained when all symbols are in RAM. DEBUG-432 takes advantage of all available Series III memory; adding more memory boards will greatly increase the probability that user symbols will all be in RAM.

The name table reuses space as it is freed by the REMOVE command. The REMOVE ALL command completely reinitializes the name table.

No Predefined Names in the DNT

There are no predefined names in the DNT. There are certain template names used as defaults when memory is displayed or examined. These names are listed in Chapter 4, both in Table 4-2, "Default Template Names" and in the section "Other Default Templates". These names should always be defined as template names. A set of default template definitions is listed in Appendix C and can be entered into the DNT by using the INCLUDE command to include the file DEB432.TEM.

Two reference names, CC and CP, are automatically entered in the DNT when the first breakpoint is serviced by the debugger and are assigned new values at every breakpoint.

Any of these names can be deleted from the DNT.

INCLUDE

Very often when debugging a program, the same set of commands is used to set up the environment at the beginning of each session. These commands can be placed in a Series III file with a text editor and executed during the debugging session using the command:

```
    INCLUDE filename [LIST]
```

where filename is any Series III filename, and the LIST option determines whether the commands in the file will be echoed at the CRT.

As mentioned above, the file DEB432.TEM contains a set of template definitions, followed by the commands BASE 16 and SUFFIX 16. Other commands may be added to this file, or another INCLUDE file may be created. For example, assume the file START.INC has the following commands:

```
    LOG test.log
    INCLUDE deb432.tem
    INIT
    DEBUG test.eod
    START
```

Then the command

```
    INCLUDE start.inc LIST
```

causes the commands in the file to be executed. The commands in this file will be echoed, since the LIST option was used; the template definitions in DEB432.TEM will not be echoed.

STARTING A DEBUGGING SESSION

The commands discussed above are usually used to initialize the System 432/670 and the debugger when you first begin a debugging session. Depending on the environment when you enter the debugger, you may use these commands in different combinations.

This section describes five likely sequences for using the environment commands:

Standard startup
Exiting and returning
Reloading 432 memory
Loading with SAVE and RESTORE
Recovering from a crash

In general, the INIT command should be given before any command that loads a System 432/670 memory image. If the user loads a new image on top of an executing one, it is highly likely that the processors will fault, requiring an INIT and then another load operation.

STANDARD STARTUP SEQUENCE

The most common sequence for starting a debugger session is shown in Figure 3-1.

```
-RUN DEB432
SERIES III 432 SYSTEMS LEVEL DEBUGGER, V1.00

?include deb432.tem          -- include the Intel-supplied templates
?init                        -- initialize the System 432/670
TOP OF MEMORY IS: 7FFFF
?debug test.eod              -- load the program to be debugged,
                                -- start polling, and enable
                                -- logical addressing

...                          -- set breakpoints, patch memory,
                                -- define templates.

?start                       -- start the system
...                          -- debug program
?exit
```

Figure 3-1. Standard Startup Sequence

EXITING AND RETURNING

It is possible to exit the debugger and then reenter it without having to reload memory. This will not work if the System 432/670 IP board has faulted or is uninitialized. Also, the last program to have interfaced to the System 432/670 through the IP board must have been the debugger.

Given these restrictions, it is possible to sign on to the debugger with the standard sequence discussed above, start the memory image, exit from the debugger, use the Series III for other purposes, and finally sign back onto the debugger and connect with the memory image. Figure 3-2 shows how to sign back on.

```
-RUN DEB432
SERIES III 432 SYSTEMS LEVEL DEBUGGER, V1.00

?include deb432.tem          -- include the templates
?debug                       -- connect to the memory image
```

Figure 3-2. Exiting and Returning

The debugger does not remember breakpoint definitions between invocations; they must be deactivated or removed before you sign off, and redefined when you sign back on.

RELOADING MEMORY

Figure 3-3 shows how to load a memory image during a debugging session. It assumes that the file DEB432.TEM, containing the standard templates, was included during the initial startup.

```
...
?init              -- reset the System 432/670 and clear memory
?debug test.eod    --  load   the   new   image   and   begin   logical
                   -- addressing
...                -- set breakpoints, patch memory, etc.
?start             -- begin debugging.
```

Figure 3-3. Reloading Memory

USING SAVE AND RESTORE

The SAVE and RESTORE commands can manipulate entire memory images; RESTORE is faster than DEBUG or LOAD for loading an image. However, to take advantage of this the user must first LOAD the image from a linked EOD and then SAVE the image in a file (the size of the static load image may be obtained from the link map), as shown in Figure 3-4.

```
-RUN DEB432
SERIES III 432 SYSTEMS LEVEL DEBUGGER, V1.00

?include deb432.tem
?init
TOP OF MEMORY IS: 7FFFF
?debug test.eod
?save !0 TO !1FFFF to :f1:test.sav      -- 1FFFF from link map
?start
...
?init
TOP OF MEMORY IS: 7FFFF
?restore :f1:test.sav
?start
...
?exit
```

Figure 3-4. Using SAVE and RESTORE

RECOVERING FROM A CRASH

Two kinds of crashes can require the System 432/670 to be reset: a process in 432 memory may have caused a crash severe enough to place the System 432/670 in a "fatal" state; or the cable connection between the Series III and System 432/670 may have come loose, causing the Interface Processor to fault. The debugger displays the message:

```
    ERR 329: FATAL IP ERROR
```

Figure 3-5 shows the sequence to recover.

Use INIT SYSTEM instead of INIT to preserve the crashed system's memory image for examination. The physical address in the command should be the address of the last byte of memory.

Note that resetting the System 432/670 resets all executing processors.

```
    ...
    ERR 329: FATAL IP ERROR
    ?init system !7ffff       -- reset 432 hardware but not memory
    ?debug                    -- restart logical addressing
    ?
```

Figure 3-5. Recovering from a Crash

CHAPTER 4
EXAMINING AND MODIFYING MEMORY

INTRODUCTION

DEBUG-432 lets users examine 432 memory locations and modify the contents of these locations. The debugger can also reference locations symbolically. This chapter covers the representation of iAPX 432 addresses, a simple introduction to templates, memory references, examination and modification of memory, and the Data Structure Table.

REPRESENTATION OF iAPX 432 ADDRESSES

The debugger views memory as either a linear array of 8-bit bytes or a collection of *segments*. The first view is called *physical memory*; the second is called *logical memory*. Physical memory is referenced through physical addresses, and logical memory through logical addresses. Full syntax for addresses is in Appendix A; a discussion of memory segments is in the *Introduction to the iAPX 432 Architecture*.

PHYSICAL ADDRESSES

A physical address is simply an ordinal number that specifies an index into the array of bytes that makes up physical memory. In the debugger, a physical address is written as an exclamation point (!) followed by an ordinal indexed from zero:

```
!12345          -- the 12,346th byte of memory
```

The ordinal can also be replaced by an expression (see "Expressions" below).

LOGICAL ADDRESSES

A logical address has the format:

```
directory index ^ segment index [{.} offset]
                                [{!}       ]
```

where each of the three components is either a non-negative integer or an expression, enclosed in parentheses, which evaluates to a non-negative integer.

The *directory* and *segment indices* are the *coordinates* of a segment; together they are called the *object reference* or an *access descriptor*. Users can obtain coordinates of all static segments in the memory image from the linker load map. (See the *Intel 432 Cross Development System VAX/VMS Host User's Guide*.)

If the separator between the object reference and the *offset* is an exclamation point (!), the offset is interpreted as a byte offset. If the separator is a period, interpretation of the *offset* component depends upon the type of segment identified by the object reference:

- If an access segment is identified, offset *n* designates the *n*+1st access descriptor of the segment.
- If a data segment is identified, offset *n* designates the *n*+1st byte of the segment.
- When setting breakpoints in an instruction segment (see Chapter 5), *n* designates the *n*+1st bit in the segment.

If the offset is omitted altogether, the address identifies the whole segment.

For example, the following are valid logical addresses:

```
14^3            -- a segment (assume it is a data segment)
                -- with directory index 14 and segment index 3.

14^3.1234       -- the 1,235th byte in that data segment.

3^27            -- a segment (assume it is an access segment) with
                -- directory index 3 and segment index 27.

3^27.12         -- the 13th access descriptor in that access
                -- segment.

3^27!12         -- the 13th byte of access segment 3^27
```

INTERCONNECT ADDRESSES

DEBUG-432 can address the iAPX 432 interconnect address space (described in the *System 432/600 System Reference Manual*). The interconnect space is examined and modified exactly like memory. If the object table entry for a logical address is an interconnect descriptor, the address is said to be a logical interconnect address; the syntax is the same as for any other logical address. A physical interconnect address is specified by an ordinal expression preceded by two exclamation points (!!):

```
!!8      -- The address of the ninth byte of interconnect space
```

This document will use the term "interconnect address" for physical interconnect addresses. Logical interconnect addresses are treated along with logical memory addresses.

TEMPLATES

The iAPX 432 architecture is highly structured; it consists of *objects*: hardware-recognized data structures. DEBUG-432 provides constructions called *templates* to examine and modify the objects in the memory image. A template describes how the data in an object is organized in memory and how that information should be displayed when the user examines the object.

A number of templates are supplied with the debugger. These describe access and segment descriptors, some standard system objects, and some 8-, 16-, and 32-bit data items. For example, a template called PROCESS_AS describes part of a Process Access Segment, another template called AD describes the coordinates of an Access Descriptor, and a template called B8 describes bytes. The Intel-supplied templates are in a file called DEB432.TEM. Table 4-1 lists the templates supplied in DEB432.TEM and gives a brief description of what each template displays. A complete list of these templates is in Appendix C.

Users may define templates for system objects or for their own data structures. System objects are described in the *iAPX 432 General Data Processor Architecture Reference Manual* and in the *iAPX a432 Interface Processor Architecture Reference Manual*. Template definition is covered in detail in Chapter 6.

In this section, we will define two templates, CONTEXT_AS and CONTEXT_DS, which describe a context access segment and a context data segment, respectively. The file DEB432.TEM supplies two templates with these names; the ones defined in this section are similiar to those two templates.

A template to describe the first 9 slots of a context access segment might be:

```
TEMPLATE context_as IS
  ctxt_ds:     @0;          -- AD to context data segment
  const:       @1;          -- AD to constants data segment
  prev:        @2;          -- AD to calling context
  msg:         @3;          -- AD to message object
  curr_ctxt:   @4;          -- AD to current context
  EAS_1:       @5;          -- AD to Entered Access Segment 1
  EAS_2:       @6;          -- AD to Entered Access Segment 2
  EAS_3:       @7;          -- AD to Entered Access Segment 3
  domain:      @8;          -- AD to defining domain
END
```

Each field is given a name (CTXT_DS, DOMAIN). The notation "@*number*", called an *access descriptor index*, describes the access descriptor whose index is given by the number. There is an optional clause that has been omitted here, called the access attribute. For example the fields PREV and CURR_CTXT could have been defined as:

```
      prev:         @2   ACCESS context_as;
      curr_ctxt:    @4   ACCESS context_as;
```

The access attribute gives the template name that is to be used to describe the object pointed to by the access descriptor. The access attribute may be present only for fields defined as an access descriptor index. In the case of the PREV and CURR_CTXT fields, access descriptors 2 and 4 both point to context access segments, so CONTEXT_AS is appropriate. The CTXT_DS field might have an access attribute of CONTEXT_DS:

```
    ctxt_ds:        @0   ACCESS context_ds;
```

This declares that the template CONTEXT_DS should be used to interpret the object pointed to by the first access descriptor in a context access segment. The use of the ACCESS attribute is discussed below in "Dot Notation."

A template to describe the first four fields of a context data segment would look like:

```
    TEMPLATE context_ds IS
      status:    [0, 16];     -- The context status
      sp:        [2, 16];     -- the operand Stack Pointer
      inst_idx:  [4, 16];     -- index of instruction segment in domain
      ip:        [6, 16];     -- the Instruction Pointer
    END
```

Again, each field has a name. The notation "[i, k]", called a bit string descriptor, selects a bit field starting at byte i that is k bits long. For example, INST_IDX describes a 16-bit field that starts at byte 4 and IP describes a 16-bit field that starts at byte 6.

Note that each numberic value displayed by a template field has its most significant bit on the left, least significant bit on the right; the debugger displays low address items first, next higher address items next, etc.

Template definition is covered in detail in Chapter 6.

Table 4-1. Templates in DEB432.TEM

Template	Displays
AD	- one access descriptor (di^si)
AS	- 2 rows of 8 ADs (64 bytes total)
B16	- 16 bits as an unsigned value
B32	- 32 bits as an unsigned value
B8	- 8 bits as an unsigned value
BS	- 32 bits as an unsigned value
CONTEXT_AS	- The first 8 ADs of a context access segment
CONTEXT_DS	- The first part of a context data segment
DESCR	- Uses a variant to select the correct template to display any object table entry
DS	- 4 rows of 8 short ordinals (64 bytes total)
EXTRACT	- 32 bits as an unsigned value (used in expressions)
FLT	- When applied to a process access segment, FLT displays the context fault area of that process
MEM	- 4 rows of 16 bytes followed by ASCII for those 16 bytes (Uses DUMP)
ORD	- An ordinal
PFLT	- When applied to a process access segment, PFLT displays the process fault area.
PROCESS_AS	- the first 10 ADs of a process access segment
PSORFLT	- When applied to a processor access segment, PSORFLT displays the processor fault area.
RAD	- An AD with the rights bits (Raw Access Descriptor)
RAS	- The first 14 RADs of a segment
SO	- A short ordinal

The following templates are used in defining those above:

CH	- an ASCII character (or "." if unprintable)
DUMP	- 16 bytes followed by the ASCII (using CH). This template is used to build the MEM template.
FREE	- a free entry in an object table (used by DESCR)
F_AREA	- generic fault area (used by FLT, PFLT, PSORFLT)
F_OR_T	- "F" or "T" depending on value of first bit
HEADER	- a header entry in an object table (used by DESCR)
INTERCONNECT	- an interconnect descriptor (used by DESCR)
PROC_STAT	- part of the process status (used by PROCESS_AS
REFINE	- a refinement descriptor (used by DESCR)
STORAGE	- A storage descriptor (used by DESCR)
SYSTEM_TYPE	- Applied to certain descriptors, SYSTEM TYPE displays the system type of the object (used by STORAGE, REFINE)
TYPE_DES	- a type descriptor (used by DESCR)

REFERENCES

A reference is a combination of an address and a template name (written as address:template). When the reference is used, we say "the template is being applied against memory." The debugger will accept just an address wherever a reference is required; the debugger will supply a default template when the address is used.

The address part of a reference determines what memory the template is applied against. This memory is called the bit stream. If the address is a physical or interconnect address, the bit stream is either the rest of memory or 64K bytes, whichever is smaller, starting at the address. If the address is a logical address, the bit stream is the rest of the segment, starting at the address.

Example References:

```
    9^9:context_as       - apply the template CONTEXT_AS to segment 9^9
    21^3.12:ord          - apply ORD to the rest of segment 21^3,
                           starting at offset 12
    !8:extract           - apply EXTRACT to memory starting at byte 8
    !!4:so               - apply SO to interconnect space, starting at
                           byte 4
```

A reference can be given a name and entered into the DNT. The type of the name is REFERENCE (i.e., the command DIR REFERENCE will display all of the references defined in the DNT). Once the name has been defined, it can be used wherever the reference may be used:

```
    name IS reference
```

For example

```
    my_process is 4^12:process_as
    tree is 7^1B:node
    psor is 1^1
```

In this last example, the reference PSOR is defined to be an address, which means that whenever PSOR is used, the debugger will select a default template to form a complete reference. With two exceptions (modifying memory and using an address in an arithmetic expression), the "Default Template Selection Algorithm", described below, is used to determine the default template. Briefly: the debugger has a table of template names for each hardware recognized system object (PROCESS_AS, CONTEXT_AS, etc.). If the address is a logical address and is the address of one of these objects, a template from that table is selected. If that template is not defined or if the logical address does not describe a system object, one of the generic template names AS or DS will be used. If the address is a physical address or interconnect address, the template DS will be always be used.

A reference expression is either a reference, a reference name, or an expression of the form:

```
    reference_expression:template_name
```

The rightmost template name in such an expression is the template that will be used to form the reference. For example:

```
3^16.431        -- Template part determined when expression used.
tree:flt        -- same as 7^1B:flt, using tree as defined above
2^2:ds:as       -- same as (2^2:ds):as which is same as 2^2:as
psor:b8         -- same as 1^1:b8, using psor as defined above
```

DOT NOTATION

A reference has a template part and, as shown in the section on "Templates", templates may have named fields. The debugger provides a dot notation, to refer to the individual template fields of a reference or reference expression. The general syntax is:

```
reference_expression [. field name]...
```

where the field name must be a field of the template used in the reference_expression. Assume that 9^1C is a context access segment (see Figure 4-1 below), then the expression:

```
9^1C:context_as.prev
```

uses dot notation to reference the PREV field. Using the CONTEXT_AS template defined in the previous section, we see that this expression is a reference to the third access descriptor (i.e., the access descriptor having index 2). Likewise, if we assume that 9^1D is a context data segment, then the expression:

```
9^1D:context_ds.ip
```

references the instruction pointer field of the context data segment.

When using dot notation, we can also use a named reference preceding the dot:

```
c is 9^1C:context_as

c.prev                  -- same as "9^1C:context_as.prev"
```

Also, as mentioned above, whenever a reference is required, just an address may be supplied; the debugger will supply a default template if one is not provided. Since we are assuming that 9^1C is a context access segment, the default template that would be selected is CONTEXT_AS. Therefore, the following expression is also acceptable and equivalent to the ones shown above:

```
9^1C.prev               -- same as "9^1C:context_as.prev"
```

TRAVERSING ACCESS DESCRIPTORS

The syntax for dot notation allows chains of .field name to appear. If a field of a template designates an access descriptor, then both the value of the field and the object that the field points to are interesting. The dot notation provides a way to traverse the access descriptor and examine the indicated object.

When the dot notation is used to traverse a field of a template, a new reference is formed. The address part of the reference is the access descriptor, taken from 432 memory, indicated by the template field to the left of the dot. The template part of the reference is either taken from the ACCESS clause of the field, or a default will be selected by the debugger (see "Default Template Selection Algorithm"). For example,

```
9^1C:context_as.prev.domain
```

traverses the PREV field of context access segment 9^1C. If we assume that the value of this field is 9^0F, another context access object, then the reference formed by "9^1C:context_as.prev" is the same as "9^0F:context_as" and the example above is the same as:

```
9^0f:context_as.domain
```

since the PREV field points to the calling context access segment of a context.

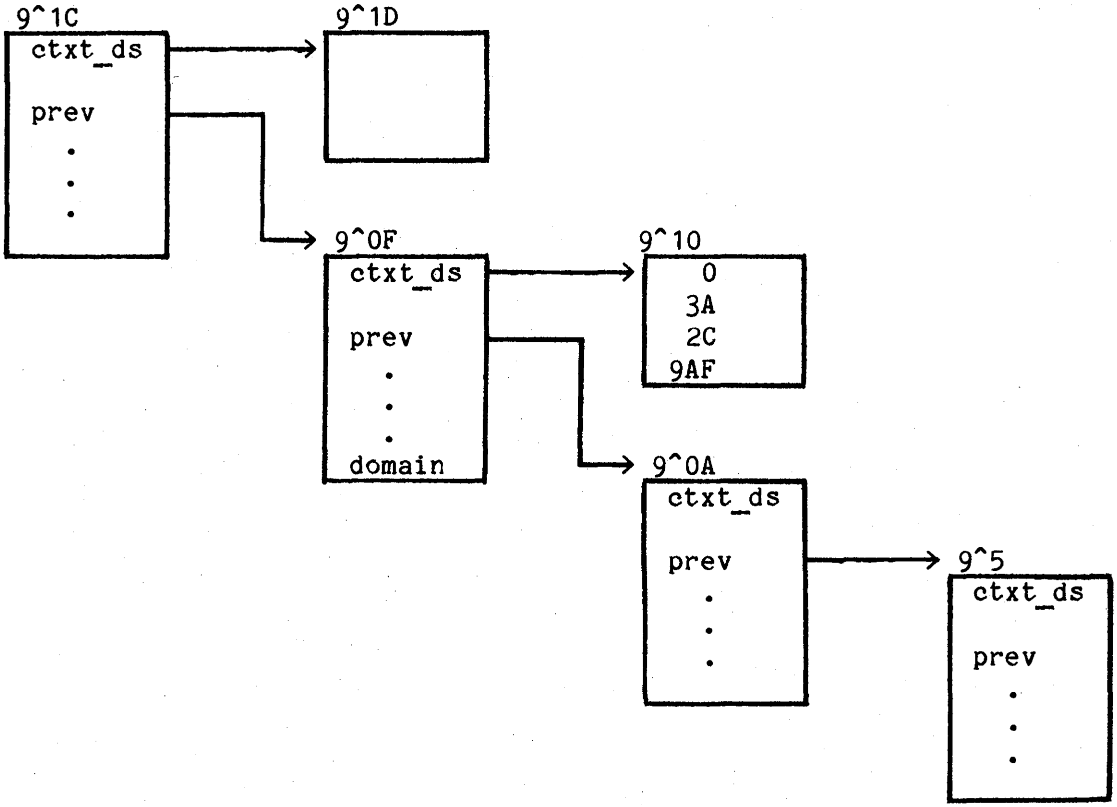


Figure 4-1. A Chain of Context Access Objects

ACCESS PATHS

When dot notation is used to access a template field of a reference, the part of the expression that precedes the rightmost dot is called the access path to the reference. In the expression

```
    9^1C:context_as.prev
```

"9^1C:context_as" is the access path to the reference 9^1C:context_as. Consider the example from above, of traversing an access descriptor:

```
    9^1C:context_as.prev.domain
```

in this example, "9^1C:context_as.prev" is the access path to the reference 9^0F:context_as.

As will be seen in the section on the "Data Structure Table", the debugger provides a way to make the template field names of one reference directly accessible, without having to retype the access path to the reference.

DEFAULT TEMPLATE SELECTION ALGORITHM

Whenever a reference (i.e., address:template) is required by the debugger, only an address need be used. The debugger will supply a default template name to make the address into a reference. In all cases but two, the debugger uses the default template selection algorithm to select a template name (the exceptions are modifying memory or using just an address in an expression, see the bottom of the next page).

For physical and interconnect addresses, the default template is DS. If the address is a logical address, the following algorithm is used for choosing a default template name:

1. If the logical address is the address of a segment (i.e., a logical address with no offset), a name is selected from the list in Table 4-2, depending on the base and system type of the segment. The debugger will display the name selected followed by a <RETURN>. If a template with the selected name is defined, that template is used as the default. Otherwise, go to step 2.

2. If the logical address has an offset or if no template is defined with the name selected in Step 1, the debugger selects either the name DS or AS as the default, corresponding to the base type of the segment.

3. If a segment has a system type other than the ones defined in Table 4-2, the debugger chooses DS or AS as the default, depending on the base type of the segment.

4. If neither AS nor DS is defined, an error occurs.

Table 4-2. Default Template Names

	Base Type	
System Type	Data Segment	Access Segment
0	Generic_DS	Generic_AS
1	Reserved_DS	Reserved_AS
2	Object_Table	Domain_AS
3	Instruction_Segment	Reserved_AS
4	Context_DS	Context_AS
5	Process_DS	Process_AS
6	Processor_DS	Processor_AS
7	Port_DS	Port_AS
8	Carrier_DS	Carrier_AS
9	SRO_DS	SRO_AS
10	Communication_Segment	Type_Def_AS
11	Descriptor_Control_DS	
12	Refinement_Control_DS	

OTHER DEFAULT TEMPLATES

These seven templates are used by the debugger as default templates in different situations. A definition for each of these appears in the file DEB432.TEM.

Template Name — Situation Where Used as a Default

1. AD - in templates: default for "@number" bit identification
2. AS - for displaying an access segment (See algorithm above)
3. B8 - used for modifying memory
4. BS - in templates: default for "[i:j,k]" bit identification
5. DESCR - default for displaying 'SD attribute logical address
6. DS - for displaying a data segment (See algorithm above)
7. EXTRACT - when a reference is used in an expression

The definition of these templates has a considerable impact on the operation of the debugger. These names, and the names of the default templates listed in Table 4-2 should be used only as template names, and care should be taken when redefining them.

EXAMINING MEMORY

To examine memory, the user enters either a reference expression followed by a repetition clause or a reference expression with dot notation indicating that a template field is to be examined.

The DEBUG-432 command for examining memory has the following syntax:

```
    source [repetition]
or
    source [. field name]...
```

where source is a reference expression, repetition is either "LEN number" or "ALL", and field name is the name of a template field.

The source reference expression is used to obtain the address:template pair for examining memory. As described in "References," the address determines the bit stream -- the memory the template will be applied against. The debugger applies the template to memory and follows the specifications in the template definition for displaying that memory.

If the reference expression does not have a template part (i.e., it is just an address), then a template is selected according to the default template selection algorithm, described above.

The repetition field specifies how many times the template is to be applied to memory. If more than one repetition is specified, the template is applied successively, each time starting where the previous application left off. This will continue until either the repetition count is complete or the end of the bit stream is reached.

If a field name is present it must be a field of the template used in the reference or ".ALL". A field name causes the display of just that field, ".ALL" causes the entire reference to be displayed indented two spaces and preceded by the access descriptor for the reference.

Sample Memory Examination Commands:

(Assume 1^1 is an access segment)

```
1^1:mem        -- display segment 1^1 using template MEM
1^1            -- display segment 1^1 using default
1^1:ds         -- display segment 1^1 using template DS

1^1.4:ad       -- using ad, display starting at 5th AD of 1^1
1^1.4:ord      -- using ord, display starting at 5th AD of 1^1

!18:descr      -- use template DESCR to display, starting at byte 18

1^1:ad len 3   -- display the first 3 access descriptors in 1^1
1^1:ad ALL     -- display all the access descriptors in 1^1
```

```
    !28:storage.base_addr     -- display the base_address field using the
                              -- template STORAGE applied to memory
                              -- starting at byte 28.

    !!4:so          -- display interconnect register 2 (bytes 4 and 5)
```

Many templates are written to display starting with the first byte, word, or access descriptor of the bit stream (an exception is the FLT template in DEB432.TEM). If the template is written to examine more memory than is defined by the address, the debugger pads the memory by adding zeros to the end. For example, in the command "1^1:mem" if segment 1^1 is 20 bytes long, but the template MEM examines 64 bytes, the last 44 bytes examined will be zeros.

Examples of Examining Memory:

All examples of memory examination are assumed to have BASE and SUFFIX settings of 16.

Example 1:

```
    ?4^1C
    PROCESS_AS
    STATUS:          0C048   [BOUND, NOT_FAULTED]
    PROC_DS:        4^ 23
    CURR_CTXT:      4^ 20
    PGLOB_AS:       4^ 1F
    LOC_OBJ_TAB:    2^  3
    PROC_CARR:      0^  0
    DISP_PORT:      4^ 1D
    SCHED_PORT:     0^  0
    FAULT_PORT:     0^  0
    CUR_MSG:        0^  0
    CUR_PORT:       0^  0
    CUR_CARR:       0^  0
    SURR_CAR:       0^  0
    ?
```

In this example, the debugger recognized the segment with coordinates 4^1C as a Process Access Segment, then used the template PROCESS_AS (from DEB432.TEM) to examine the data in this object and display it in tabular form.

Example 2:

```
    ?5^4.3
    GENERIC_DS
        0      345      27      0      0      0      0      0
       23       3F     456      0      0      0      0      0
        0        0       0      0      0      0      0      0
        0        0       0      0      0      0      0      0
    ?
```

In this example, the reference used to examine memory is "5^4.3:DS". The debugger has interpreted the address 5^4.3 as the address of a data segment, of system type "GENERIC DS". No template named GENERIC_DS was found in the debuggers name table, so the template DS was used by default.

Example 3:

```
    ?3^2.23:AD
    1^3
    ?
```

In this example, the user has told the debugger to interpret the 4 bytes starting at location 3^2.23 as an access descriptor.

Example 4:

```
    ?3^2.23:ORD
    1F003F
    ?
```

In this example, the user has told the debugger to examine the same location as in Example 3, but this time to interpret the data as a 32-bit quantity. The template ORD accomplishes this interpretation.

Example 5:

```
    ?1^1:DS
       46F      5F       0      0     45F     5F     33F     5F
        2F      2F     3FF     5F     21F     6F     43F     5F
         0       0       0      0       0      0       0      0
       19F      6F       0      0       0      0     41F     5F
```

This example uses the template DS, which displays the contents of a segment as a sequence of double bytes, to display the segment 1^1. To display just the first 32 bits of the segment the template ORD (for ordinal) may be used:

```
    ?1^1:ord
    5F046F
```

The same segment can be displayed using the template AS, which displays the contents of the segment as a series of access descriptors:

Example 6:

```
    ?1^1:as
    5^ 46   0^  0   5^ 45   5^ 33   2^  2   5^ 3F   6^ 21   5^ 43
    0^  0   0^  0   0^  0   0^  0   6^ 19   0^  0   0^  0   5^ 41
```

To display just the first two access descriptors, the template AD may be used (and the repetition clause):

```
?1^1:AD  len 2
  5^ 46
  0^  0
```

Finally, we can examine the segment using the template MEM, which displays the contents of the segment as both bytes and ASCII characters:

Example 7:

```
?1^1:mem
 6F  4 5F  0  0  0  0  0 5F  4 5F  0 3F  3 5F 0  'o._....._._.?._.'
 2F  0 2F  0 FF  3 5F  0 1F  2 6F  0 3F  4 5F 0  '/./..._...o.?._.'
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0  '................'
 9F  1 6F  0  0  0  0  0  0  0  0  0 1F  4 5F 0  '..o..........._.'
```

MODIFYING MEMORY

Memory is modified by assigning a particular value, given by an expression, to a reference. The DEBUG-432 command for modifying memory has the following general format:

destination [. field name]... := expression

where destination is a reference expression and field name must be a field of the template used for the destination. The value of the expression can be either an integer or a reference. If the expression is a reference, data will be copied from the reference on the right-hand side of the := to the reference on the left-hand side, from low address to high address, four bytes at a time.

If the reference_expression for the destination does not have a template part, the default template is B8. If the expression on the right-hand side is a reference without a template name, the default template is the template being used on the left-hand side.

The memory that will be modified is determined in two different ways, depending on the presence or absence of the field name. If the field name is present, only the memory indicated by that field is modified (more precisely, the memory indicated by the bit identification field, see Chapter 6).

If the field_name is not present, the template is used to determine how much of the bit stream is to be modified. The debugger applies the template to memory as if it were examining memory, but the display is suppressed. The debugger keeps track of the highest bit in the bit stream that is used (or "touched") by the template. This is called the high water mark of the template application. The part of the bit stream that will be modified starts at the given address and goes through the high water mark.

CAUTION

When the destination of the memory modification command is just a reference expression (i.e., with no dot notation), ALL the memory from the address of the reference to the high water mark will be modified.

If the right-hand expression is an arithmetic value, it is truncated or extended with zeros to fit. If it is a memory reference, data is moved four bytes at a time, from low address to high address, until the left-hand side is filled. The template on the right-hand side is used to identify what memory will be copied, exactly as the template on the left-hand side is used to identify what memory will be modified. If the right-hand side memory is not as much as is required by the left-hand side, the high order bits written into the destination will all be zero.

Examples

(Assume BASE and SUFFIX are 16)

```
?1^1:ord                    -- first examine the memory
5F046F
?1^1:so := OFFFF            -- now modify the low double byte
?1^1:ord                    -- now re-examine to see the change
5FFFFF

?10^3.3:AD                  -- First examine the memory
 OB^  8
?10^3.3:AD := OAF003F       -- replace the 4th access descriptor in
                            -- access segment 10^3 with the descriptor
                            -- whose hex value is OAF003F.
?10^3.3:B32                 -- display the result
OAF003F
?10^3.3:AD                  -- now show it as an access descriptor
 OA^  3
?

?6^4:B32 := 7^4B            -- replace the first 32 bits of segment
                            -- 6^4 with the first 32 bits of segment
                            -- 7^4B (the default template on the
                            -- right is also B32).

?6^4:CONTEXT_DS.IP := 9AF-- replace the ip field of the context_ds
                            -- when applied to segment 6^4 with 9AF.
                            -- this will modify bytes 6 and 7 of 6^4.

?5^2 := 8A                  -- replace the first byte of 5^2 with 8A
?                           -- (the default template is B8)
```

```
?5^5:B32 := 0FFFF_FFFF     -- fill the data segment 5^5 or the part
?5^5.4:DS := 5^5           -- specified by DS with binary ones
```

In the last example, the template DS determines how much of segment 5^5 gets filled with ones. DS touches up to 64 bytes. When it is applied on the left-hand side to 5^5.4, it specifies bytes 4 through 67, provided 5^5 is at least 68 bytes in length. If 5^5 is shorter than 68 bytes, the entire data segment is filled with ones.

DATA STRUCTURE TABLE

To permit the fields of an object to be conveniently referenced, the debugger supports the Data Structure Table (DST). Using a variation on the dot notation described earlier in this chapter, the user can cause the template field names of a reference to be entered into the DST. This table of names is searched by the debugger if it cannot find a definition for a name in the DNT.

For the rest of this section, we use the templates CONTEXT_AS and CONTEXT_DS, defined in the section on "Templates" earlier in this chapter. The chain of context objects is pictured in Figure 4-1.

We begin with an example: Assume that 9^1C is a context access segment and we want to change the IP value in the context data segment of 9^1C's caller to 8AB if it is not already:

```
?9^1C:context_as.prev.ctxt_ds.ip
9AF
?9^1C:context_as.prev.ctxt_ds.ip := 8AB
```

(Recall PREV is a field name in CONTEXT_AS pointing to the caller of a context, CTXT_DS points to the context data segment of a context.)

To avoid retyping the access path to the context data segment, we invade the reference by typing the access path to the object followed by a dot:

```
?9^1C:context_as.prev.ctxt_ds.
ctxt_status:  0
sp:          3A
inst_idx:    2C
ip:         9AF
?
```

As a side effect of the invasion, the invaded object is displayed (using the CONTEXT_DS template defined in this chapter):

Now the individual fields of the object can be referenced without typing the entire access path:

```
    ?ip                  -- Check the value of IP (is it 8AB?)
    9AF
    ?ip := 8AB           -- Change the value to 8AB
    ?
```

INVASION

Invasion uses a variation of dot notation to cause the template field names of the invaded reference to be entered into the DST.

Any access path may be invaded. This has the effect of entering all of the field names of the reference's template into the DST. These names can now be used unqualified. There are three different syntaxes for doing an invasion:

1. With a trailing dot or ^ (if ^ is used, the reference being invaded is not displayed):

   ```
   ?9^1C:context_as.prev.ctxt_ds.
   ```

   or

   ```
   ?9^1C:context_as.prev.ctxt_ds^
   ```

2. A dot on a command line by itself will invade the expression most recently used to examine memory and display the invaded reference:

   ```
   ?9^1C:context_as.prev.ctxt_ds
     9^ 10
   ?.                       -- This is the invading dot
   ctxt_status:   0
   sp:           3A
   inst_idx:     2C
   ip:          9AF
   ?
   ```

3. A dot preceding a field name causes the same behavior as in item 2 above, except that the display is suppressed for the reference being invaded:

   ```
   ?9^1C:context_as.prev.ctxt_ds
     9^ 10
   ?.ip
   9AF                           -- just the field is displayed
   ```

THE CURRENT ACCESS PATH

When the user invades a reference, the debugger keeps track of the current access path to the reference (i.e., the access path preceding the invading dot or caret). In addition to invasion, the debugger supports three commands that operate on the current access path:

BACK - removes the rightmost element from the access path, effectively undoing the most recent invasion

OUT - tells the debugger to "forget" about the current access path

PATH - displays the current access path onto the CRT.

The following gives an example of the state of the current access path:

```
Command                  Access Path after Command

?9^1C:context_as^        9^1C:context_as
?prev
  9^ 0F                      "
?prev^                   9^1C:context_as.prev(9^0F)
?prev
  9^ 0A                      "
?prev^                   9^1C:context_as.prev.prev(9^0A)
?prev
  9^  5                      "
?back                    9^1C:context_as.prev(9^0F)
?prev
  9^ 0A                      "
?out                     no access path
```

EXPRESSIONS

An expression can be an explicitly specified ordinal value such as 4 or 345F, an identifier whose value is a positive or negative integer, a value referenced from 432 memory, or any arithmetic combination of these three. The debugger supports the arithmetic operations listed below. These operations are given in order of lowest to hightest priority, with operators of the same priority grouped together. The order of execution within operators of the same priority is from left to right; order of execution can be overridden using parentheses.

```
+     --addition
-     --subtraction (two's complement)

-     --unary minus

*     --multiplication
/     --division
REM   --remainder
MOD   --modulus

**    --exponentiation
```

Since unary minus (-) is considered an operator, a negative integer may be entered wherever an expression is legal.

For example:

```
i:INTEGER := -20
param:B32 := -255
(-2)**expo
```

If the result of an expression has the high order bit set to 1, the debugger displays the value as both a 32-bit ordinal and a 32-bit integer:

```
?0FFFF_FFFFE + 1
-1       0FFFFFFFF

?-1
-1       0FFFFFFFF
```

REM and MOD behave as they do in Ada. (See the <u>Reference Manual for the Ada Programming Language</u>.)

An expression entered at a prompt will be evaluated by the debugger and the result displayed on the console.

<u>Examples</u>:

```
!(23+54)          -- the physical address !77
```

The following three examples show the operations commonly used to calculate the index and EAS selector of an access selector (for example, the access selector 2B):

```
2B/4             -- index:  divide by 4 (i.e., shift right by 2 bits)

2B rem 4         -- EAS selector:  the selector REM 4

5^3.(2B/4)       -- Use the index part of an access selector to index
                 -- into a segment (presumably one of the entry access
                 -- segments)
```

INTEGERS: NAMING AN EXPRESSION

The debugger permits a name to be given to an expression and entered into the DNT. The type of such a name is called INTEGER. The syntax for declaring an INTEGER is:

```
    name: INTEGER := expression
```

The value of the expression must be a 32-bit integer and this number is the value given to name. These names can be used anywhere a number may be used in an expression. However, names may not be used without surrounding parentheses to form a logical address (assume a and b are INTEGERS):

```
    a^b
```

is not allowed, although

```
    (a)^(b)
```

and

```
    !a
```

are acceptable.

REFERENCES IN EXPRESSIONS

A reference without a field name may be used in an expression. The reference is interpreted just like the reference on the left-hand side of a memory modification command. That is, the template is applied to memory to calculate the high water mark. The bits that will be used in the expression are the bits from the address of the reference through the high water mark, unless there are more than 32 bits, in which case only the low order 32 bits are used. If the reference has no template part, the default template EXTRACT is used, rather than B8. DEB432.TEM defines EXTRACT as a 16-bit quantity. For example:

```
    1^1!4:b8 + 5
```

forms the sum of 5 and the 8 bits from byte 4 of segment 1^1. On the other hand:

```
    1^1!4 + 5
```

forms the sum of 5 and the 16 bits starting at byte 4 of segment 1^1.

FIELDS IN EXPRESSIONS

The template field of a reference may be used in an expression, just as a reference may be used in an expression. The bits that are identified by the template field (more precisely, by the bit identification component of the field. See Chapter 6) are the bits used in the expression.

As an example, the following expression will find the current instruction access index of the context object (from the context data segment) and display it divided by 4 (i.e., display just the index part):

```
    9^1C:context_as.ctxt_ds.inst_idx/4
```

INTRODUCTION

The iAPX 432 architecture supports the concepts of process and context (i.e., an activation of a procedure). DEBUG-432 can monitor the dynamic behavior of particular processes in a multiprocess environment and particular contexts associated with a procedure. This chapter covers these features:

- Examining the set of processes at breakpoints; selecting a process to be the current process; restarting a process (or processes).
- Setting, displaying, activating, deactivating, and removing breakpoints.
- Manipulating the call stack of contexts; selecting the current context.

PROCESSES

When the first breakpoint of the debugging session occurs, the reference variables CP and CC are defined. The initial value of CP is the access descriptor of the process which first reached a breakpoint; the value of CC is the address of the breakpointed context. Each process that reaches a breakpoint is added to the set of breakpointed processes, called the process set. When a process is RESUMEd, it is removed from the process set.

The debugger supports three basic operations on this set:

- EXAMINE the contents of the process_set
- SELECT a process from the process_set to become the current process
- RESUME a process (i.e., send it to a GDP for execution)

BREAKPOINTS

Breakpoints are key elements in debugging. They let the user halt execution of a process ("break") at pre-selected positions. The debugger supports three types of breakpoints:

1. Instruction offset breakpoints. The process or processes break when a particular instruction in an instruction segment is reached.

2. Procedure entry and exit breakpoints. The process or processes break when a call is made to (or a return is made from) a particular instruction segment or any of the instructions segments in a particular domain.

3. Event breakpoints. The process or processes break when one of the following events occurs: 1) a new instruction is executed, 2) a call or intersegment branch is executed, 3) a return is executed, or 4) a fault occurs. Event breakpoints are classified according to the kind of events they track.

DEBUG-432 implements breakpoints with two different mechanisms. Instruction offset breakpoints are implemented using the fault mechanism of the iAPX 432 architecture. DEBUG-432 places an illegal class code in the instruction segment at the requested offset. Each process that executes this instruction will fault. These illegal class faults are handled by the portion of DEBUG-432 that resides in 432 memory, and are announced as breakpoints.

All other breakpoints are implemented using the iAPX 432 tracing facility. A process may be in one of four trace modes: none, full, flow, or fault. When a trace event occurs, it is handled by the portion of the DEBUG-432 that resides in 432 memory and is announced as a breakpoint. Event breakpoints that track faults use fault trace mode. Event breakpoints that cause a break before each instruction require full trace mode. Procedure entry and exit breakpoints, and call or return event breakpoints use flow trace mode. Since the process may be in only one trace mode at a time, these three sets of breakpoints are mutually exclusive. Instruction offset breakpoints, which do not use the tracing mechanism, may be active at the same time as any other breakpoint type.

The process trace mode can be changed only while the process is not executing. To ensure that this condition is met, breakpoints that use the trace facility can be set only when the process is at a breakpoint. Instruction offset breakpoints, however, may be set at any time -- before the process has begun to execute, while it is executing, or after it has broken.

Whenever a break event occurs (whether through the fault or trace mechanism), the process begins executing DEBUG-432 code. Thus the current context of the breakpointed process (CP.CURR_CTXT) is NOT the breakpointed context. The debugger, however, sets its own reference variable CC (current context) to the value of the breakpointed context.

In summary, the debugger enforces these rules:

- Only instruction offset breakpoints may be set for ALL processes, without explicitly giving a list of processes.

- Only instruction offset breakpoints may be set while the process is running.

- Instruction trace, fault trace, and flow trace breakpoints are mutually exclusive.

DEBUG-432 provides commands to set, display, activate, and deactivate breakpoints.

SETTING BREAKPOINTS

The general form of a breakpoint definition is:

```
[name:] break [OF process list]
```

where break may be:

```
BA instruction
BE {inst seg ¦ domain}
BX {inst seg ¦ domain}
BO {CALL ¦ RET ¦ INST ¦ FAULT}
```

The process list field can contain several process specifications, separated by commas. For BA breakpoints the process list may be ALL, indicating all processes.

See the command descriptions for BA, BE, BO, and BX for a further discussion of each type. The address fields in these commands, instruction, inst seg, domain, and process list are logical addresses. The address may be entered as an access descriptor and optional offset, or a name whose value is a logical address.

For example,

```
?BA 2^3.456 OF 4^28             -- set breakpoint at offset 456 of
?                               -- instruction segment 2^3 for the
                                -- process whose Process Access
                                -- Segment is at 4^28
?proc IS 4^28:PROCESS_AS        -- name the process at 4^28 proc
?BA 2^3.456 OF proc             -- sets same breakpoint as above
?

?BA 14^21.171 OF 4^44, 16^21    -- multiple process breakpoint that
?                               -- sets a breakpoint at 14^21.171
                                -- for both 4^44 and 16^21.
```

If, after at least one process is at a breakpoint, the process field is omitted, the debugger uses the current process (CP), which is described below (see "Processes"). Thus,

```
?BA 4^74.311
```

is the same as

```
?BA 4^74.311 OF CP
```

NAMING BREAKPOINTS

Every breakpoint must have a name. The user can specify a name defining the breakpoint, or can let DEBUG-432 assign a breakpoint name.

To name a breakpoint, simply precede the set-breakpoint command with a name. For example:

```
?mainbreak: BA 2^3.456 OF ALL

?stackbreak: BE 1^2 OF 4^44

?coffeebreak: BX 1^2
```

If the user does not name a breakpoint, the debugger will name it. The default name assigned by the debugger will be "B<ddd>", where <ddd> is one, two, or three decimal digits; for example, B23 or B4. The debugger will not attempt to use a name already in use.

Every breakpoint will be entered into the DNT. The type of the name will be BREAK (i.e., the command DIR BREAK will list all the breakpoints currently defined). The breakpoint will have either the attribute "Active" or the attribute "Inactive" (see below, "Activating And Deactivating Breakpoints").

Breakpoints may be deleted from the DNT with the REMOVE command. For example,

```
?REMOVE b0          --removes breakpoint b0

?REMOVE break       --removes all breakpoints
```

ACTIVATING AND DEACTIVATING BREAKPOINTS

When the user sets a breakpoint, it is "active." When a process (for which the breakpoint is set) reaches the breakpoint, it stops executing user code and is added to the breakpointed process set (see "Processes"). The user may deactivate a breakpoint: the breakpoint is lifted (removed from 432 memory) but the definition is still in the DNT. The form of the deactivate command is:

```
DEACTIVATE [name ¦ ALL]
```

Deactivated breakpoints can be reactivated, unless the breakpoint has been removed from the DNT. The syntax for the ACTIVATE command is:

```
    ACTIVATE [name ¦ ALL]
```

The directory of breakpoints in Figure 5-1 contains both active and inactive breakpoints. Inactive breakpoints are indicated by a "*" to the left of the breakpoint definition. An active breakpoint with a process at the breakpoint is indicated by a "-" to the left of the definition; other active breakpoints have nothing preceding the definition.

```
?dir break
B0          BREAK    BA 9^47.50 of 9^16
B1         -BREAK    BA 9^49.50 of 9^2b
B2         -BREAK    BA 9^4B.50 OF 9^40
B3         *BREAK    BO INST OF 9^16
B5          BREAK    BO CALL OF 9^16
?
```

Figure 5-1. Sample list of breakpoints in the DNT

The LOAD and DEBUG commands deactivate all breakpoints.

REACHING A BREAKPOINT

When a process reaches a breakpoint, it stops executing the user's code and is added to the process set. If the debugger is waiting for input, the breakpoint is announced immediately. If the user is typing a command or entering input to the 432 program, the debugger sounds the bell (i.e., CONTROL-G) and waits until the command or input line is completed before announcing the breakpoint.

The breakpoint is announced as follows:

name. break identifier: address OF process

where break identifier is characteristic of the breakpoint type, name is the name of the breakpoint, address designates the logical address of the breakpoint, and process identifies the process that reached the breakpoint.

The address displayed when a breakpoint is announced is normally the location of the instruction to be executed when the process is RESUMEd. For BX breakpoints, however, the address may be greater than the bit offset of the return instruction by as much as 32 bits.

The following break_identifiers are used by the debugger for display:

```
BREAK AT        -- instruction offset breakpoints

BO INST         -- event breakpoints
BO CALL
BO RET
BO FAULT

BREAK ENTER     -- enter and exit breakpoints
BREAK ENTER (DOM: address)
BREAK EXIT
BREAK EXIT (DOM: address)

FAULT AT        -- exceptional event announcements
TRACE AT
```

The FAULT break identifier occurs when the debugger detects a user context level fault. The TRACE break identifier occurs when a trace event happens and the debugger was not expecting one. For example, if the "Exiting and Returning" sequence in Chapter 3 is used, any breakpoints which were active at the end of the session still remain in the load image. When the user signs back on, the debugger no longer has the definitions of those breakpoints. If a process reaches such a breakpoint, it is announced with FAULT AT or TRACE AT.

If a fault is announced, the user may correct the condition causing the fault and attempt to reexecute the faulting instruction using the RESUME command.

Recall that when an instruction offset (BA) breakpoint is set, ANY process which reaches the specified instruction will break. If the process which broke was NOT in the process list used to define the break, and the ALL option was not used, then the breakpoint will be announced as above, with the message "N.B. NOT THE DEFINING PROCESS".

Some examples:

```
mainbreak. BREAK AT: 2^3.456 OF 4^44

stackbreak. BREAK ENTER: 1^2.121  OF 4^44

coffeebreak. BREAK EXIT: 1^2.235 OF 4^44
```

At any time the only processes that the user can be sure are stopped are the ones in the breakpointed process_set. The rest of the system is still executing. To stop the entire system, the IPC command may be used:

```
IPC ALL,2
```

CONTEXTS AND THE CALL STACK

Associated with each process is a chain of one or more context objects (see the iAPX 432 GDP Architecture Reference Manual for a discussion of context objects). This chain makes up the call stack associated with that process. The call stack describes the sequence of activations that have led to the currently breakpointed procedure. For example, if the three procedures A, B, and C are declared in an Ada program, and A has called B, which in turn has called C, then the current call stack is organized as follows:

```
A
B
C
```

C is called the BOTM of the stack, A is the TOP of the stack, B is UP from C, and B is DOWN from A. The current context is associated with procedure C.

The commands UP, DOWN, TOP, and BOTM affect only the value of the variable CC. TOP, for example, sets CC to the initial context of the process (in our example, the context associated with procedure A). BOTM moves the current context back to the context where the breakpoint occurred (in our example, the context associated with C). UP and DOWN have obvious meanings.

The STACK command displays the chain of contexts, and the return instruction address associated with each context. For example:

```
?STACK
CONTEXT              INSTRUCTION
8^0C                 7^6D.43F
8^8                  7^70.26A
8^5                  7^74.127
8^1                  7^7B.1A9
```

In this example, 8^0C is the bottom of the stack, the last called context. 8^1 is the top of the stack.

Each breakpointed process has a "current context" associated with it. When the process first reaches a breakpoint, that "current context" is the breakpointed context. This "current context" value may be changed by UP, DOWN, TOP, and BOTM whenever the process is the current process (CP). When the SELECT command is used to change CP, the debugger saves the value of CC for the old CP before selecting a new CP. The EXAMINE command displays the process access descriptor and this saved value of "current context" for each process at a breakpoint. If an old CP is reSELECTed, CC will be set to this saved value, not the breakpointed context. The STACK command, however, displays the call stack beginning with the breakpointed context, regardless of the value of the current context.

INTRODUCTION

DEBUG-432 provides a special mechanism to examine and modify 432 objects. The mechanism is based on templates, structures that define the parts of interest of the objects. This chapter describes how to define templates and use them to access objects.

As discussed in Chapter 4, 432 memory examination and memory modification both use "References," which are defined to be an address:template pair.

A template is a list of one or more field definitions. Each field describes components of an object. For example, the template PROCESS_AS (from the file DEB432.TEM) consists of a set of field definitions that describe some of the access descriptors in a Process Access Segment. (Appendix C shows the definitions of the entire contents of DEB432.TEM; consult this list for the definition of PROCESS_AS.)

Templates have two major roles in the debugger: identifying the pieces of data that make up an object and displaying the data.

- Memory examination uses identification and display.

- Memory modification consists of identifying the memory to be modified and then copying new values into this memory.

A template is somewhat like a cookie cutter. It defines a pattern in a linear stream of bits just as a cookie cutter defines a pattern in the dough it is applied against. The "dough" for templates is a bit stream starting at an address. If the address is a logical address, the bit stream is the rest of the segment, starting at the logical address. If the address is a physical or interconnect address, the bit stream is the 64K bytes starting at the address. To avoid confusion with other bit fields, the bit stream that the template is being applied against will be called the bit stream throughout the remainder of this chapter, while other fields will be called "bit strings."

TEMPLATE DEFINITION

A template is composed of zero or more fields followed by an optional variant part. The basic template definition syntax is

```
    TEMPLATE name IS
      field [;
      field] ...
     [variant part]
    END
```

with an arbitrary number of fields specified. The template name is used to form References, the variant part of a template is discussed at the end of this section. A template field has three optional components:

```
    [field name:[:]] [bit identification] [IS display format]
```

Since all three components are optional, the field itself is optional. In the template syntax above, this means a semicolon (;) may appear without a field after it.

The field name identifies the field. It may be displayed as part of the output generated by the template, or used to traverse an access discriptor, or to select a field for display.

The bit identification clause specifies which part of the "bit stream" this field is to use.

The display format clause of a field guides the debugger when a template is being used by giving a list of editing and conversion specifications. A conversion specification tells the debugger to take the bit string, identified by the bit identification clause of this field, and convert it to characters for display on the CRT. If the display format clause is omitted, a default format is used.

HOW A TEMPLATE WORKS

When the template part of a reference is used to display memory, each field is interpretted as follows:

The field name is printed if it is followed by a single colon (:).

The display format is interpreted as a sequence of conversion and/or editing specifications, from left to right. Editing specifications cause formatting characters or text to be displayed. Each conversion specification in a display format causes the bit identification specification to be used to identify a bit string for conversion. The converted string is displayed. Some bit identification forms cause a bit string with an absolute position in the "bit stream" to be used (e.g., the eighth byte); other forms identify the bit string relative to the last identification (e.g., the next byte).

The variant part allows a value to be identified in memory and used to select one of many alternatives. Each alternative can contain a variant part.

For example, consider a very simple template containing one field:

```
    ?TEMPLATE bite IS
    ??  a_byte: [0, 8] IS 16U;
    ??END
```

The field name is A_BYTE. The bit identification clause is [0, 8]. This particular bit identification clause is called a bit string descriptor. It describes a string of 8 bits starting at byte zero of the "bit stream" to which the template BITE is applied. The display format clause contains only the conversion specification "16U", which tells the debugger to display the identified bit string as a base-16, unsigned integer. Notice that the semicolon (;) following the 16U could have been omitted.

As shown, the debugger prompts with a double question mark (??) after the first line of a template definition, indicating that the template definition command is not complete. When the END is reached, the debugger returns to a single question mark (?) prompt.

Template definitions may be entered in free format, so the entire definition may be placed on a single line:

```
    ?TEMPLATE bite IS  a_byte: [0,8] IS 16U  END
```

This template may be used to examine 432 memory (using segment 1^1 shown in Figure 6-1, below):

```
    ?1^1:bite
    a_byte: 6F
```

OVERVIEW OF A FIELD

Recall that a template field has three optional components:

```
    [field name:]  [bit identification]  [IS display format]
```

Field Name

The field name identifies the field. When the template is used to display data at the terminal, the field name, if one exists, will appear immediately before the data for that field. The name is left justified and blank-filled so that all the field names in the output generated by the template occupy the same number of columns. The name is followed by a colon (:). The user can suppress the display of the field name by using a double colon (::) instead of a single colon (:) after the field name in the template definition (such a fieldname can be used to traverse if associated with an access descriptor).

Bit Identification

The syntax for a bit identification clause is:

```
                  { bit string descriptor
    [@number.]... { access descriptor index
                  { template name
```

A bit string descriptor gives the starting address and length of a bit string. An access descriptor index identifies an entire access descriptor by giving its index. A template name identifies a bit string by calling on another template to perform the identification.

The optional @number provides for dereferencing access descriptors. It permits the bit identification clause to identify an access descriptor and then, in the segment pointed to by the rightmost access descriptor, to identify a bit string.

Display Format

The display format part of a template field is a list of display elements:

```
    display element [, display element] ...
```

The display elements are evaluated from left to right. A display element has the syntax:

```
                      { integer
                      { enumeration
                      { ASCII
    [<repeat count>]  { template name
                      { new line
                      { string
                      { blanks
                      { "["display format"]"
```

The angle brackets (<>) are required to delimit the repeat count. Square brackets ([]) indicate that the count is optional. The square brackets shown in quotes, however, are part of the syntax.

A display element is either a conversion or an editing specification. A conversion specification tells the debugger to convert a bit string, identified by the bit identification clause, into characters to be displayed at the console. There are four kinds of conversion specifications: integers, enumeration, ASCII, and a template name.

The editing specifications of a display format permit text and special characters to be output. There are three kinds of editing specifications: text strings, blanks, and new line.

In addition, a repeat count may prefix any specification. This provides a method for writing repetitive formats compactly. Square brackets can be used to enclose a series of specifications so that the entire group can be repeated.

```
?1^1:ds
    46F     5F       0       0      45F     5F      33F     5F
     2F     2F     3FF      5F      21F     6F      43F     5F
      0      0       0       0        0      0        0      0
    19F     6F       0       0        0      0      41F     5F
```

This example uses the template DS, which displays the contents of a segment as a sequence of double bytes, to display the segment 1^1. To display just the first 32 bits of the segment the template ORD (for ordinal) may be used:

```
?1^1:ord        --displays most significant non-zero digit on left
5F046F
```

The same segment can be displayed using the template AS, which displays the contents of the segment as a series of access descriptors:

```
?1^1:as
5^ 46    0^  0    5^ 45    5^ 33    2^  2    5^ 3F    6^ 21    5^ 43
0^  0    0^  0    0^  0    0^  0    6^ 19    0^  0    0^  0    5^ 41
```

To display just the first two access descriptors, the template AD may be used (and the repetition clause):

```
?1^1:AD  LENGTH 2
  5^ 46
  0^  0
```

Using the template MEM the segment can be displayed as corresponding bytes and ASCII characters, low address to high address is left to right, top to bottom::

```
?1^1:mem
 6F  4 5F  0  0  0  0  0 5F  4 5F  0 3F  3 5F 0  'o._....._._.?._.'
 2F  0 2F  0 FF  3 5F  0 1F  2 6F  0 3F  4 5F 0  '/./..._...o.?._.'
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0  '................'
 9F  1 6F  0  0  0  0  0  0  0  0  0 1F  4 5F 0  '..o..........._.'
```

Figure 6-1. Sample Bit Stream.

Different templates displaying the same stream of bits. This same bit stream will be used in examples throughout this chapter.

BIT IDENTIFICATION AND DISPLAY FORMAT SPECIFICATIONS

The rest of this section defines and demonstrates the various bit identification and display format specifications. We omit the debugger prompts when showing template definitions. Since the display and identification functions of a template work hand in hand, they will be described together, in parallel. Most of the examples use the segment 1^1, shown in Figure 6-1. The template BITE, defined earlier, is a good starting point:

```
TEMPLATE bite IS
  a_byte: [0, 8] IS 16U;
END
```

Recall that the template displays the first byte of segment 1^1 as:

```
?1^1:bite
a_byte: 6F
```

CAUTION

The template definitions in this chapter assume that the templates are to be entered with the current SUFFIX setting 10. However, the examples of using templates to examine memory assume that thereafter the current BASE and SUFFIX settings are 16. This is consistent with typical debugger use. As a matter of fact, INCLUDing DEB432.TEM sets BASE 16 and SUFFIX 16 in its last 2 lines.

TEXT STRINGS (DISPLAY FORMAT)

A text string is the simplest editing specification that can be introduced into a display format. A text string is any string of characters surrounded by quote marks. Quote marks can be put into the string by putting two quote marks in a row. For example:

```
TEMPLATE bite IS
  a_byte: [0, 8] IS "before", 16U, "after";
END
```

"before" and "after" are text strings. This template will now display:

```
?1^1:bite
a_byte: before6Fafter
```

The debugger does not put any characters into the display unless explicitly requested by the display format. Therefore, blanks should be added to clean up the display:

```
TEMPLATE bite IS
  a_byte: [0, 8] IS "before ", 16U, " after";
END
```

```
?1^1:bite
a_byte: before 6F after
```

INTEGER (DISPLAY FORMAT)

An example of an integer display format is "16U". The general form is either:

```
        baseU [:width]
```

or

```
        baseS [:width]
```

Base indicates the desired numeric output base, and is a decimal number in the range 2..16 or zero (e.g., 0U); zero requests that the current setting of the BASE command be used.

U and S stand for "unsigned" and "signed", respectively. An unsigned conversion specification treats the identified bit string as an ordinal value greater than or equal to zero. A signed conversion treats the identified bit string as a twos-complement integer. For example, the sign extended 3-bit bit string 100 would be converted to "4" under 10U and to -4 under 10S. The bit string 0100 would be converted to "4" by both 10U and 10S.

The optional width is the minimum decimal number of character positions the converted bit string should occupy when it is displayed as a number The maximum width is 64. If the number occupies fewer characters than the width, it is right justified and blank filled. If it is too wide, the specified width is ignored. For example:

```
    TEMPLATE bite IS
      a_byte: [0, 8] IS 10U:2, 16U:3, 2U:12
    END
```

would display as:

```
    ?1^1:bite
    a_byte: 10#111# 6F  2#1101111#
```

The base 10 number was too large to fit into a field of width 2 so the width was ignored. The pound signs (#) each occupy one character position. So do a minus sign in a negative number and the leading zero in a number whose first digit would otherwise be in the range A..F. The 6F was right justified and blank filled to occupy three characters. This accounts for the blank preceeding the 6F. The 2#1101111# was also right justified and blank filled to occupy twelve characters, accounting for the two blanks preceeding it.

This example also demonstrates that if the output base specified for display of an integer differs from the current setting of the BASE command, the integer is displayed in the form base#number#. If displayed in the default output base, the integer is displayed alone. Minus signs preceed the base: 0F displayed via "[0,4] is 15S" displays as -15#10#.

BIT STRING DESCRIPTORS (BIT IDENTIFICATION)

The bit string descriptor is the most direct form of identifing a bit string. The general format is:

```
    "["byte start  [:bit start],  bit length"]"
```

where the square brackets ([]) shown in quotes are actually part of the descriptor (but the quotes are not!). A bit string descriptor explicitly specifies the starting position and length of a bit string. The starting position is given with a byte offset (byte start), optionally followed by a bit offset from the beginning of that byte (bit start). If the bit start is omitted, it defaults to zero. The length of the bit string is given in bits (bit length). The starting position is relative to the beginning of the "bit stream" that the template is being applied against (for a picture of the "bit stream" see the discussion of By_P and Bi_P, below). The following example shows the AD template supplied in the file DEB432.TEM (recall template definitions assume SUFFIX 10):

```
    TEMPLATE ad IS
      [2:4, 12] IS OU:3, "^";          -- Directory index and caret
      [0:4, 12] IS OU:3;               -- Segment index
    END
```

This template displays one access descriptor, with the directory index first, followed by a caret, and then the segment index (the rights bits are ignored). For example (examples assume BASE 16):

```
    ?1^1:ad
      5^ 46
    ?                                  -- next debugger prompt
```

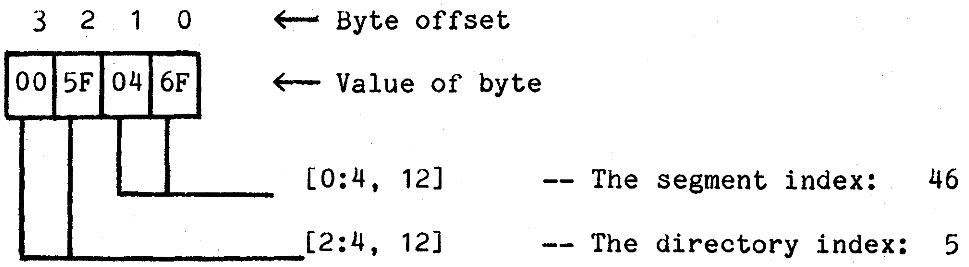
To verify that the AD template is working, we can define another template, ORD (also from DEB432.TEM), to display a 32-bit quantity:

```
    TEMPLATE ord IS
      [0, 32] IS OU                    -- (32 is assumed decimal)
    END
```

and then display the same 32 bits as the AD template displayed:

```
    ?1^1:ord
    5F046F
    ?                                  -- next debugger prompt
```

A picture of this 32 bit value tells us that the AD template is working:


3 2 1 0
← Byte offset
00 5F 04 6F
← Value of byte
[0:4, 12] -- The segment index: 46
[2:4, 12] -- The directory index: 5

NEW LINE (DISPLAY FORMAT)

The AD template demonstrates a multiple field template whose entire display is on one line.

The new line editing specification, a slash (/), tells the debugger to output a <CR><LF> pair to the CRT, positioning the cursor at the beginning of the next line.

As mentioned earlier, the debugger displays only characters explicitly requested by the template. However, once the entire template is finished displaying, the debugger will output a <CR><LF> if necessary to position the debugger's prompt at the beginning of a line. The last two examples, displaying the first 32 bits of 1^1 as AD and as an ORD, illustrate this rule. Although neither the AD nor the ORD template contains a new line character, the debugger inserted a <CR><LF> pair after the template was finished, so that the next debugger prompt (?) was at the beginning of a line.

REPEAT COUNT AND GROUPING BRACKETS (DISPLAY FORMAT)

A repeat count may prefix any specification in the display format. The repeat count is written as an integer expression (value) in angle brackets, immediately preceding the specification to be repeated. Square grouping brackets may be used to collect a series of specifications together, so that the entire series can be repeated as a whole. For example:

```
TEMPLATE dashes IS
  [0, 8] IS <10>"-", OU, <5>"*";                -- (SUFFIX 10 assumed)
END
```

will display the first byte of 1^1 as:

```
?1^1:dashes
----------6F*****
```

The repeat count may also be used in front of a conversion specification, to cause memory to be displayed multiple times:

```
TEMPLATE bytes IS
  [0,8] IS <16>Ou
END
```

this template displays sixteen bytes:

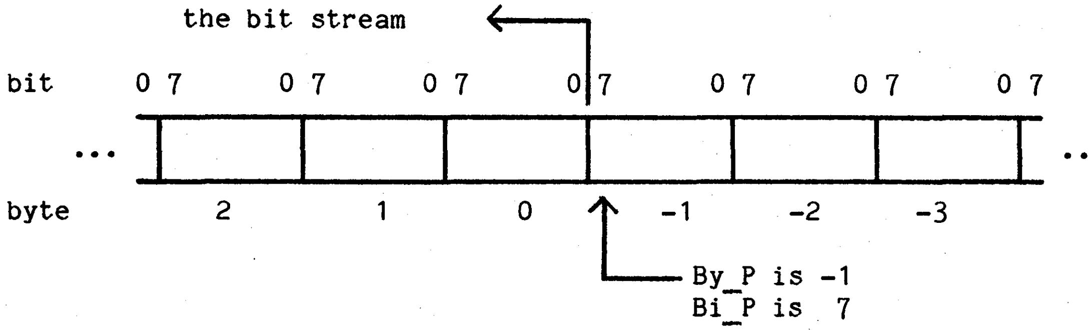
```
?1^1:bytes
6F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6F
```

However, the same byte is displayed all sixteen times, with no spaces between the repetitions. Spaces can easily be introduced either by grouping a blank with the integer conversion or by including a display width on the integer conversion itself.

The variables By_P and Bi_P provide a way for a conversion specification to be repeated and cause each repetition to convert a different bit string.

BY_P AND BI_P (BIT IDENTIFICATION)

By_P and Bi_P may be used in a bit string descriptor together with a repeated conversion specification to extract a sequence of values from memory. The debugger maintains the two pseudo variables, By_P and Bi P, for each template while it is being applied to memory. These two variables are similiar to local procedure variables in Ada: every template application has its own local copy of By_P and Bi_P. By_P always points to the most recent byte of the bit stream that the template has referenced and Bi_P points to the highest order, most recently referenced bit of that byte. These two values are always normalized so that Bi_P is less than 8. Since the first byte of the bit stream is viewed as byte zero by a template, the following picture illustrates the initial values of By_P and Bi_P.



The debugger issues an error if an attempt is made to identify bits that precede the beginning of the "bit stream".

Using By_P and Bi_P the template BYTES, from above, may be rewritten to display successive bytes of memory with blanks between them:

```
TEMPLATE gulps IS
  [By_P:Bi_P+1, 8] IS <16>0U:3;              -- (16 assumed decimal)
END

?1^1:gulps                                        -- (Suffix 16 NOW)
 6F  4 5F  0  0  0  0  0 5F  4 5F  0 3F  3 5F 0
```

The first byte identified was [-1:7+1, 8] which, when normalized, is the same as [0:0, 8]. After the first byte is identified, By_P is updated from -1 to 0. Bi_P will be 7, the last bit of byte 0 that was referenced. In the above example, the successive values of the pair By P and Bi_P, just before each byte was identified were: (-1:7), (0:7), (1:7), ..., (14:7).

The template, GULPS, can be modified to display multiple rows of 16 bytes by grouping the display of one row together with a new line. This group can then be repeated to display successive rows:

```
TEMPLATE gulps IS
  [by_p:bi_p+1, 8] IS <4>[<16>0U:3, /]        -- (base 10 assumed)
END

?1^1:gulps                                    -- (base 16 assumed)
 6F  4 5F  0  0  0  0  0 5F  4 5F  0 3F  3 5F  0
 2F  0 2F  0 FF  3 5F  0 1F  2 6F  0 3F  4 5F  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 9F  1 6F  0  0  0  0  0  0  0  0  0 1F  4 5F  0
```

This is the technique used for the templates DS, AS, and MEM, shown in Figure 6-1.

EXPRESSIONS IN TEMPLATES

Four positions in a template definition can be defined as expressions, instead of just numbers: byte start, bit start, bit length, and repeat counts. This was demonstrated above with the GULPS template, which had a bit start of "Bi_P+1". The permissible expressions may have the arithemetic operators +, -, *, and /, where * and / have higher priority than + and -. An operand used with these operators may be a number, an integer identifier in the DNT, By_P, Bi_P, or a parenthesized expression.

BLANKS (DISPLAY FORMAT)

Blanks are introduced into a display format with the notation:

```
countX
```

where count is the number of blanks that should be displayed when the template is applied. This notation is just an abbreviation for that number of blanks inside of quotes or a single blank preceded by a repeat count. For instance, 10X is the same as <10>" ".

ENUMERATIONS (DISPLAY FORMAT)

Enumerations are used to map bit strings into identifiers or characters. The form of an enumeration is quite similiar to an enumeration type definition in Ada:

```
( [number =>] element, [number =>] element, ... )
```

Here an element is either an identifier or a string in quotes. If any element in an enumeration is preceded by a number, they all must be preceded by a number. If numbers are not present, the first element is displayed if the identified bit string has value 0, the second element if the string has value 1, and so on.

If numbers are present, they must be ordered, smallest first (leftmost), largest last (rightmost).

Like all of the conversion specifications, an enumeration specifies how to convert a bit string from binary to characters to display at the terminal. For example, the enumeration

```
    (alpha, beta, gamma, theta)
```

uses the default to specify that a bit string should be evaluated and different identifiers displayed depending on the value of that bit string. If the string has value 0, the identifier alpha should be displayed, if 1 then beta, if 2 then gamma, if 3 then theta. If the bit string has a value greater than 3 the debugger will issue an error message. A "sparse" conversion can be given by explicitly specifying which bit string values map into which elements. For example:

```
    (1=>"0", 2=>"1", 4=>"2", 8=>"3", 16=>"4", 32=>"5", 64=>"6")
```

If we assume a 6-bit field contains a number that is an exact power of two, this enumeration converts that 6-bit string into the character representing the log (base 2) for the bit string and then displays the character at the terminal.

Using enumerations, we can construct a template which will display the letters "T" or "F" instead of the bit values 1 or 0 for a byte:

```
    TEMPLATE bits IS
      [by_p:bi_p+1, 1] IS <8>(F, T);
    END

    ?1^1:bits
    TTTTFTTF
```

At first glance this is puzzling, since previously we have displayed the first byte of segment 1^1 and its value is 6F. The reason for this apparent discrepancy is that the bits are being displayed low order to the left and high order to the right by the repetitive enumeration. Each time the enumeration is repeated, it requests a bit string be identified and displayed. The values that [by_p:bi_p+1] will assume (i.e., the bits that will be identified) are, in order, [0:0], [0:1], ..., [0:7], displayed left to right.

TEMPLATE NAMES IN DISPLAY FORMATS

A <u>template name</u> in the display format makes it possible to use already defined templates to display parts (or all) of the bit stream. A very compelling example is the template, AD. This template is used to display a single access descriptor. By using AD in a display format, we can define the template AS, used to display the first 64 bytes of a segment as access desciptors (recall that template definitions are shown under the assumption that the current SUFFIX setting is 10):

```
    TEMPLATE AS IS
      [by_p:bi_p+1, 32] IS <2>[<8>[ad, 2x], /]
    END
```

As already shown in the introduction to this chapter, AS displays segment 1^1 as:

```
    ?1^1:as
    5^ 46   0^  0   5^ 45   5^ 33   2^  2   5^ 3F   6^ 21   5^ 43
    0^  0   0^  0   0^  0   0^  0   6^ 19   0^  0   0^  0   5^ 41
```

A template used in a display format behaves as if it were being used in a debugger command (e.g., 1^1:as): the template is applied against a "bit stream" and characters are displayed at the terminal. The "bit stream" of the display format template is the bit string identified by the bit identification clause of the field (one template's bit string is another template's "bit stream"). In the above example, the template AD is successively applied against the bit strings (in decimal) [0,32], [4,32], [8,32], ..., [60,32].

When a template name appears in the display format of a field, the debugger does not check to see if the template name is actually defined until the template that contains the display format is used. For example, the template AS could be defined before the template AD, even though AS uses the template AD. However, if AS is used before AD is defined, the debugger will issue an error when it tries to use AD.

TEMPLATE NAMES IN BIT IDENTIFICATIONS

A template name may also be used as the bit identification part of a field. The template name may appear in two ways:

template name

or

"["byte start [:bit start], template name"]"

where the square brackets shown in quotes are actually part of the bit identification (but the quotes are not).

The first alternative is shorthand for "[By_P:Bi_P+1, template_name]".

The start of the identified bit string is either By_P:Bi_P+1 or byte_start:bit_start in the bit stream, depending on which alternative is used to specify the bit identification.

The length of the identified bit string is determined by applying the named template to the remainder of the "bit stream", starting at either By_P:Bi_P+1 or at byte_start:bit_start. The length is given by the formula:

```
    max_By_P * 8 + max_Bi_P + 1
```

where max_By_P and max_Bi_P represent the values for By_P and Bi_P when By_P:Bi_P was at its highest point (i.e., the high water mark) in the "inner" template. For example:

```
    TEMPLATE bite IS
      [0, 8] IS 0U
    END

    TEMPLATE dbyte IS
      lo: bite  IS 0u;       -- same as   lo: [By_P:Bi_P+1, bite] is 0u;
      hi: bite  IS 0u;       -- same as   hi: [By_P:Bi_P+1, bite] is 0u;
    END
```

In this example BITE in the LO field identifies the bit string [0, 8] (which leaves By_p:Bi_P at 0:7). In the HI field, BITE identifies the bit string [1, 8]. The bit length is the same for both fields, since the high water mark for BITE when it is applied is always By_P:Bi_P = 0:7, which when entered into the above formula yields:

```
    0 * 8 + 7 + 1  = 8
```

ACCESS DESCRIPTOR INDEX (BIT IDENTIFICATION)

In this method of identifying a bit string, the user specifies the index of an access descriptor after an at sign (@). The index must be a non-negative number and may be followed by an ACCESS clause:

```
    @number [ACCESS template name]
```

(Recall that the @number may be preceded by zero or more occurrences of @number., as shown in the syntax earlier. This use of @number is discussed below in the section on "Dereferencing Access Descriptors".

The ACCESS clause gives the name of the template to be used in displaying the segment that this access descriptor points to. (See "Data Structure Table" in Chapter 4.) The access descriptor index, "@number", identifies the same bits as the bit string descriptor:

```
    [number * 4, 32]
```

However, only the access descriptor index may have the ACCESS clause, and the default display format for an access descriptor index is different from the default for a bit string descriptor. A typical example of the access descriptor index method for identifying bit strings is at the end of the next section.

DEFAULT DISPLAY FORMATS

If only the bit identification field of a template is provided, the debugger uses a default display format to display the identified bit string. There is a default for each of the three different bit identifications (assume foo is a template name and i, j, and k are numbers):

Bit Identification	Default Display Format
[i:j, k]	IS bs, /;
@i	IS ad, /;
foo [i:j, foo]	IS foo, /;

Note that the default display format includes a new line.

In the case of the bit string descriptor (shown as [i:j, k]), the default display is the template BS. The supplied definition of BS in DEB432.TEM is

```
TEMPLATE bs IS
  [0, 32] IS 0u;
END
```

which simply displays the low 32 bits of the identified bit string as an unsigned number in the current output base.

For an access descriptor index, the default is the template AD, which is discussed above.

When a template name is used as the bit identification, the default display format is also that template name.

The following example shows the template CONTEXT_AS, which is in the file DEB432.TEM. This template displays the first eight access descriptors of a 432 context access segment, giving their names:

```
TEMPLATE CONTEXT_AS IS
 CTXT_DS:   @0;
 CONST_DS:  @1;
 PREV:      @2;
 MSG:       @3;
 CURR_CTXT: @4;
 EAS_1:     @5;
 EAS_2:     @6;
 EAS_3:     @7;
 DOMAIN:    @8;
END
```

This template uses the default display for access descriptors "IS AD, /" to get the desired display of one field per line on the terminal screen. The next example shows the template being applied to the debugger maintained variable, CC, the current context of the current process:

```
?cc:context_as
CTXT_DS:     8^  9
CONST_DS:    7^0B8
PREV:        8^  5
MSG:         8^  7
CURR_CTXT:   8^  8
EAS_1:       0^  0
EAS_2:       0^  0
EAS_3:       0^  0
DOMAIN:      7^  2
```

ASCII (DISPLAY FORMAT)

ASCII is another conversion specification. If the keyword, ASCII, is used in a display format, the low order 7 bits of the identified bit string are converted as if ASCII were an enumeration with the following definition:

```
(nul,    soh,   stx,   etx,      eot,   enq,   ack,   bel,
  bs,    ht,    lf,    vt,       ff,    cr,    so,    si,
  dle,   dc1,   dc2,   dc3,      dc4,   nak,   syn,   etb,
  can,   em,    sub,   esc,      fs,    gs,    rs,    us,

  " ",   "!",   """",  "#",      "$",   "%",   "&",   "'",
  "(",   ")",   "*",   "+",      ",",   "-",   ".",   "/",
  "0",   "1",   "2",   "3",      "4",   "5",   "6",   "7",
  "8",   "9",   ":",   ";",      "<",   "=",   ">",   "?",

  "@",   "A",   "B",   "C",      "D",   "E",   "F",   "G",
  "H",   "I",   "J",   "K",      "L",   "M",   "N",   "O",
  "P",   "Q",   "R",   "S",      "T",   "U",   "V",   "W",
  "X",   "Y",   "Z",   "[",      "\",   "]",   "^",   "_",

  "`",   "a",   "b",   "c",      "d",   "e",   "f",   "g",
  "h",   "i",   "j",   "k",      "l",   "m",   "n",   "o",
  "p",   "q",   "r",   "s",      "t",   "u",   "v",   "w",
  "x",   "y",   "z",   "{",      "¦",   "}",   del,   tilda)
```

The ASCII conversion specification converts a byte of memory into a character for display. A template that will display ten ASCII characters surrounded by quotes would be:

```
TEMPLATE a10 IS
  [by_p:bi_p+1, 8] IS "'", <10>ascii, "'"
END
```

If this template is applied to the segment 1^1, we see:

```
?1^1:a10
'oEOT_NULNULNULNULNUL_EOT'
```

The template MEM, used in an example at the beginning of this chapter, shows the segment 1^1 being displayed as both binary and ASCII. However, MEM displays the non-printing characters as periods (.) whereas ASCII displays the name of the character. The section on "The Variant Part of a Template" shows how to create a template that displays the non-printing characters as periods and the rest of the characters as themselves.

A more pleasing example of the A10 template is when it is applied to memory containing only printing ASCII characters. For example, assume that segment 5^4 contains the following bytes:

offset:	10	9	8	7	6	5	4	3	2	1	0
value:	2E	50	52	4F	42	20	4C	45	54	4E	49

Then A10 displays:

```
?5^4:a10
'INTEL CORP'
```

DEREFERENCING ACCESS DESCRIPTORS (BIT IDENTIFICATION)

The ability to dereference an access descriptor lets the user display 432 objects that are represented by a collection of segments. The syntax for a bit identification is:

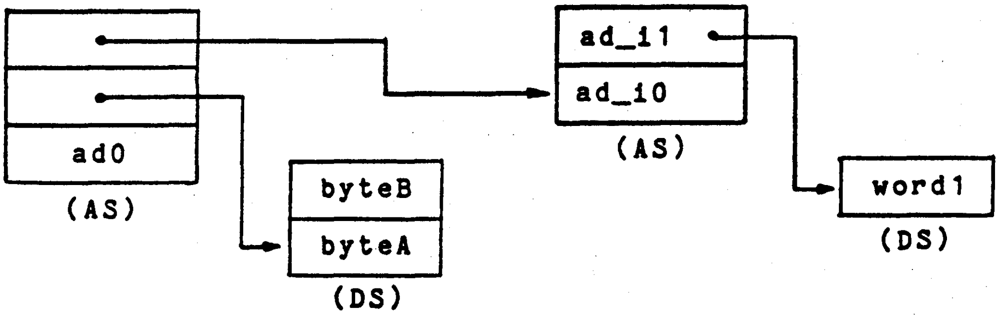
```
                  { bit string descriptor
[@number.]...     { access descriptor index
                  { template name
```

As an example, consider the following templates:

```
TEMPLATE so IS
  [0, 16] IS 0u;          -- a short ordinal
END

TEMPLATE object IS
 ad0:     @0;           -- 1st AD
 byteA:   @1.[0, 8];    -- byte 0 of segment pointed to by 2nd AD
 byteB:   @1.[1, 8];    -- byte 1 of    "       "    "  "  "  "
 ad_i0:   @2.@0;        -- 1st AD in AS pointed to by 3rd AD
 ad_i1:   @2.@1;        -- 2nd AD in "      "    "  "   "  "
 word1:   @2.@1.so;     -- 1st 16 bits of segment pointed to by
END                     --    2nd AD in AS pointed to by 3rd AD
```

The template OBJECT describes a complex object; the structure of the object is shown in Figure 6-2.



(AS) = Access Segment
(DS) = Data Segment

F-0276

Figure 6-2. Dereferencing Access Descriptors.

Each box with a name in it represents an identified bit string.

A simpler example is the template PROCESS_AS, which displays part of a process access segment. Part of the process status is displayed as well by dereferencing the process data segment access descriptor in the process access segment:

```
TEMPLATE proc_stat IS                    -- display part of status
  [0, 16]  IS 0u, 3x;
  [0,  1]  IS "[" (bound, not_bound), 2x;
  [0:1,1]  IS     (not_faulted, faulted), "]";
END

TEMPLATE process_as IS
  status:      @0.[36, 16]  IS proc_stat, /;
  proc_ds:     @0;
  curr_ctxt:   @1;
  pglob_as:    @2;
  loc_obj_tab: @3;
  proc_carr:   @4;
  dispatch_pt: @5;
  schedule_pt: @6;
  fault_pt:    @7;
  cur_msg:     @8;
  cur_port:    @9;
  cur_carr:    @10;
  surr_carr:   @11;
END
```

The field STATUS of the PROCESS_AS template dereferences the access descriptor for the process data segment. The field then identifies the 16 bits that are the process status (bytes 36 and 37). This field is displayed using the template PROC_STAT, which first displays all 16 bits and then uses enumerations to display the two lowest order bits symbolically.

VARIANT PART OF A TEMPLATE

The variant part of a template provides a method of specifying conditional display, depending on the values of the memory being displayed. The definition of the variant part of a template is patterned after the definition of the variant part of an Ada record. For ease of specification, we will rewrite the definition of a template:

```
TEMPLATE id IS
  [component list]
END
```

The component list gives the syntax for an arbitrary number of template fields optionally followed by a variant:

```
field  [; field]...  [variant part]
```

Although this syntax shows that the first field is required, recall that the definition of a field has three parts, all optional. Therefore, a template may consist of just a variant part.

The variant part of a template is defined as:

```
    CASE bit identification IS
      [WHEN choice [| choice]... =>
         [component list] ]...
    END CASE
```

The bit identification is the same as the bit identification part of a field (in a variant, it may be referred to as the discriminant). Since the last field of a component_list may be a variant, the variant part of a template may contain nested variants. The three possibilities for the choice are:

```
    number                    -- a single value
    number .. number          -- a range of values
    OTHERS                    -- all values not mentioned elsewhere
```

If OTHERS is used, it must be in the last WHEN clause of the variant. When the template is applied to memory, the discriminant is identified and, if necessary, padded to the left with zeros to make it a 32-bit number. This number is then compared to the list of choices in each WHEN clause. If a WHEN clause is found that has a choice matching the discriminant, the component list associated with that WHEN clause is applied to memory. The values for By_P and Bi_P at the beginning of this component list will be the last bit of the discriminant.

As an example, variants can provide a template that behaves just like the ASCII conversion specification, except that it displays a period (.) instead of the names of the non-printing characters:

```
    TEMPLATE ch IS
      CASE [0, 8] IS
        WHEN 16#20#..16#7D# =>
          [0, 8] IS ASCII
        WHEN OTHERS =>
                 IS ".";              -- notice: no bit identification.
      END CASE
    END
```

The template CH may be used in a display format where the specification ASCII might have been used. When the byte to be displayed is in the range of printable characters (16#20#..16#7D#), it is converted to ASCII. Otherwise, a period is displayed. Notice that to display a period, no bit identification is necessary, so the field has only the "IS display format" clause.

A WHEN clause can have many choices, strung together using the vertical bar (which separates the alternatives). For example:

```
    WHEN 5 | 10..20 | 25 =>
```

This WHEN clause matches the discriminant if the discriminant is 5 or the range 10 to 20 (inclusive) or 25.

CHAPTER 7
DEBUG-432 COMMANDS

This chapter presents the syntax and semantics of all DEBUG-432 commands. The commands are arranged alphabetically for easy reference. For a list of commands grouped by function, see the debugger reference card at the end of this manual.

ACTIVATE

Activate a breakpoint.

Syntax:

```
    ?ACTIVATE {name ¦ ALL}
```

where name is the name of a previously defined, inactive breakpoint.

Semantics:

When the user sets a breakpoint it is "active". That is, any process that reaches the breakpoint stops executing user code and is added to the breakpointed process set. A breakpoint may be deactivated explicitly using the DEACTIVATE command. All breakpoints are deactivated by the commands INIT, LOAD, DEBUG and RESTORE. The definition of the deactivated breakpoint remains in the DNT but no longer causes processes to break. To reactivate a breakpoint type:

```
    ACTIVATE name
```

where name is the name of a breakpoint. To activate all inactive breakpoints, type:

```
    ACTIVATE ALL
```

The debugger will activate breakpoints until the limit of 32 active breakpoints is reached.

Trace breakpoints (i.e., BE, BO, and BX) may only be activated if every process for which the breakpoint is defined is currently at a breakpoint.

Activating an active breakpoint has no effect.

Also see BA, BE, BO, BX, DEACTIVATE, REMOVE, and Chapter 5.

Example:

```
    ?dir stackbreak
    STACKBREAK      *BREAK    BE 1^2 OF 4^44
    ?ACTIVATE stackbreak
    ?DIR stackbreak
    STACKBREAK       BREAK    BE 1^2 OF 4^44
    ?
```

Notice that the status of the breakpoint was inactive, as indicated a the "*" to the left of the definition.

BA

Set an instruction offset breakpoint. (Break At a 432 instruction.)

Syntax:

```
    [name:]BA instruction [OF {process list ¦ ALL}]
```

where instruction is the logical address of an instruction, including object access descriptor and bit offset within the segment, and process list is a set of logical addresses of process access segments, separated by commas. The default value for this field is CP.

Semantics:

This command defines a breakpoint, stores it in the DNT, and sets a breakpoint by writing an illegal class code in the instruction segment at the offset indicated. The offset component is a bit offset instead of the usual byte offset. If no offset is specified, the initial instruction displacement from the header of the instruction segment is used.

BA breakpoints are compatible with all other breakpoint types. They may be set at any time, before the process has started, while it is running, or when it is at a breakpoint.

Every process reaching the instruction in the BA breakpoint faults and is announced as a breakpoint. Therefore, if the process will be dynamically created, but does not yet exist, use "ALL" in the process list field.

Also see ACTIVATE, BE, BO, BX, DEACTIVATE, REMOVE, and Chapter 5.

Examples:

```
    ?line_20: BA 4^34.240     --sets a breakpoint in instruction
    ?                         --segment 4^34 at bit offset 240, for the
                              --current process

    ?BA 4^34 OF ALL           --sets a breakpoint at the first
    ?                         --instruction in instruction segment 4^34,
                              --and gives it a default name
```

BACK

Go back one element in the access path.

Syntax:

```
    BACK
```

Semantics:

The BACK command removes the last item from the access path, changing the current access environment (i.e., the contents of the DST are changed).

If is only one item is in the path, the BACK command will issue an error message and the access path will remain unchanged.

Also see, PATH, OUT, and the discussion of the "Data Structure Table", in Chapter 4.

Example:

```
    ?PATH
    (9^16:context_as).prev.prev.prev.prev(9^7)
    ?BACK
    ?PATH
    (9^16:context_as).prev.prev.prev(9^0B)


    ?PATH
    (9^16:context_as)
    BACK
    ERR 180: CANNOT GO BACK ANY FURTHER
    ?PATH
    (9^16:context_as)


    ?PATH
    ...(9^12:context_as)
    BACK
    ERR 180: CANNOT GO BACK ANY FURTHER
    ?PATH
    ...(9^12:context_as)
```

The three dots (...) shown in the last sequence indicate that the access path at one time was longer than eight items, the maximum length remembered by the debugger. The example shows an attempt to go BACK past the last remembered item, which is not possible.

BASE

Change the debugger output base.

Syntax:

```
    BASE [nn]
```

where nn is the desired output base in decimal.

Semantics:

The debugger's default base for numeric I/O is initially 10. The SUFFIX and BASE commands let you set or display the default input and output bases.

The INCLUDE file DEB432.TEM has a BASE command at the end to change the output base to 16.

BASE controls the numeric output base. To set it, give the BASE command with the optional value nn. The value may be a decimal number in the range 2..16, or a number in the form base#number#, where base is the desired base in base 10 and number is the desired number in that base. The decimal equivalent of the number must be in the range 2..16.

To display the current output base, type BASE without a number. The debugger always displays the base as a decimal number, regardless of the current value of the output base.

If, through a template, the debugger is instructed to output a number in a base other than the current setting of BASE, it writes the number in the form base#number#.

Also see SUFFIX, "Integer (Display Format)" in Chapter 6, and Chapter 2.

Example:

```
    ?BASE 16            -- Change the output base to hexadecimal.
    ?BASE               -- Display the current output base.
    16


    ?BASE
    10
    ?INIT
    TOP OF MEMORY IS: 524287
    ?BASE 16
    ?INIT
    TOP OF MEMORY IS: 7FFFF
```

BE

Set a procedure entry breakpoint. (Break on Entry to an instruction segment.)

Syntax:

```
    [name:]BE {inst seg | domain} [OF process list]
```

inst seg and domain are the logical addresses of an instruction segment and a domain access segment, respectively. process list is a set of logical addresses of process access segments, separated by commas. The default value of process list is "CP".

Semantics:

This command defines a breakpoint, stores it in the DNT, and sets a breakpoint. If the form:

```
    BE inst seg
```

is used, the breakpoint is triggered by an entry to a single instruction segment. If the form

```
    BE domain
```

is used, the breakpoint occurs on entry to any instruction segment in the domain.

A BE instuction is implemented by placing the process in "flow trace" mode. This implies that a BE breakpoint may not be set for a process that has a BO INST or BO FAULT breakpoint active. A BE breakpoint can only be set when the processes given in the process list are at breakpoints.

Also see ACTIVATE, BA, BO, BX, DEACTIVATE, REMOVE, and Chapter 5.

Examples:

Assume that 10^0b is the address of a domain. Then

```
    ?BE 10^0b         --sets a breakpoint at entry to any instruction
    ?                 --segment in 10^0b, for the current process.
```

Assume that 4^2b is an instruction segment, 7^10 is a process access segment. Then

```
    ?incr is 4^2b             --names the instruction segment
    ?my_process is 7^10       --names the process
    ?BE incr OF my_process    --sets a breakpoint at entry to 4^2b for
    ?                         --process 7^10.
```

BO

Set an event breakpoint. (Break On event.)

Syntax:

```
[name:]BO {INST | CALL | RET | FAULT} [OF process list]
```

where process list is a set of logical addresses of process access segments, separated by commas. The default for this field is "CP".

Semantics:

BO INST causes a process to break before every instruction. This requires that the process be in full trace mode. For any single process, only BA breakpoints may be active at the same time as a BO INST breakpoint. The instruction at the offset given when a BO INST breakpoint is announced is the instruction that will be executed next (when a RESUME command is given).

BO FAULT causes a process to break after every fault. The process must be in fault trace mode; as with BO INST, only BA breakpoints may be active when BO FAULT is set. The instruction at the offset given when a BO FAULT is announced is the instruction that caused the fault. The process will begin execution with this instruction if a RESUME command is given.

BO CALL causes a process to break after every call (or intersegment branch). This uses flow trace mode, and is incompatible with both BO INST and BO FAULT. The instruction at the offset when a BO CALL is announced is the first instruction at the destination of the call (or intersegment branch).

BO RET causes a process to break before every return instruction. This also uses flow trace mode, and is incompatible with BO INST and BO FAULT. The bit offset which is announced when a BO RET breakpoint is reached may be as much as 32 bits greater than the actual return instruction offset.

Event breakpoints may only be set for processes at a breakpoint.

Also see ACTIVATE, BA, BE, BX, DEACTIVATE, REMOVE, and Chapter 5.

Example:

```
?BO INST            --set breakpoint after each instruction in
?                   --process CP

?BO CALL OF 9^2b    --trace all call instructions for process 9^2b
?BO RET OF 9^2b     --trace all returns for process 9^2b
?
```

BOTM

Set CC to the breakpointed context of CP.

Syntax:

```
    BOTM
```

Semantics:

Whenever the current process (CP) reaches a breakpoint, the debugger sets the reference variable CC to the address of the breakpointed context. This value may be changed by the commands BOTM, DOWN, TOP, and UP. The value of CC is set to 0^0 when the current process is RESUMEd.

```
    ?BOTM
```

resets CC to the value of the breakpointed context.

To find the current value of CC, type:

```
    ?DIR CC
```

To display the object addressed by CC, type:

```
    ?CC
```

If more than one process is at a breakpoint, the user may SELECT a new current process (CP). Before changing to a new CP, the debugger saves the current value of CC: if the current CP is re-selected in the future, CC will be set to this saved value, not the breakpointed context.

Also see DOWN, SELECT, STACK, TOP, UP, and Chapter 5.

Example:

```
    ?DIR CC             --assume there was a previous "UP" command
    CC IS 10^8
    ?BOTM; DIR CC       --this will display the value of the
    ?                   --breakpointed context
    CC IS 10^0C
    ?STACK              --this prints the entire call stack
    CONTEXT          INSTRUCTION
    10^0C            7^53.50
    10^8             9^47.26B
    10^5             9^7.127
    10^1             9^0E.1A9
    ?
```

BX

Set a procedure exit breakpoint. (Set a Break on eXit from an instruction segment.)

```
    [name:]BX {inst seg ¦ domain} [OF process list]
```

inst seg and domain are the logical addresses of an instruction segment and a domain access segment, respectively. process list is a set of logical addresses of process access segments, separated by commas. The default value of process list is "CP".

Semantics:

This command defines a breakpoint, stores it in the DNT, and sets it for each process in the list.

```
    BX inst seg
```

causes the process(es) to break on return from the instruction segment, and

```
    BX domain
```

causes a break on return from every instruction segment in domain.

The breakpoint occurs before the return is executed; however, the address which is displayed when the breakpoint is announced (and by the STACK command) may be greater than the bit offset of the return instruction by as much as 32 bits.

BX breakpoints require the process to be in flow trace mode; they are incompatible with BO INST and BO FAULT breakpoints. A BX breakpoint may be set whenever the process(es) in process list are waiting at a breakpoint.

Also see ACTIVATE, BA, BE, BO, DEACTIVATE, REMOVE, and Chapter 5.

Example:

```
    ?DONE: BX 10^0B              --sets a breakpoint before return
    ?                            --from instruction segment 10^0b.
```

DEACTIVATE

Deactivate a breakpoint.

Syntax:

```
    ?DEACTIVATE {name | ALL}
```

where name is the name of an active breakpoint.

Semantics:

This command lifts a breakpoint from 432 memory but saves its definition in the DNT. The breakpoint may be reset (activated) using the ACTIVATE command. To lift a single breakpoint, type:

```
    DEACTIVATE name
```

To lift all active breakpoints, type:

```
    DEACTIVATE ALL
```

The REMOVE command will lift an active breakpoint and delete the definition of an active or inactive breakpoint from the DNT.

A trace breakpoint (i.e., BO, BE, and BX) may not be deactivated unless every process for which the breakpoint is set is currently at a breakpoint.

Also see ACTIVATE, BA, BE, BO, BX, REMOVE, and Chapter 5.

Example:

```
    ?dir stackbreak
    STACKBREAK       BREAK    BE 1^2 OF 4^44
    ?DEACTIVATE stackbreak
    ?DIR stackbreak
    STACKBREAK      *BREAK    BE 1^2 OF 4^44
    ?
```

Notice that the status of the breakpoint is inactive, as indicated by the "*" to the left of the definition.

DEBUG

Enable logical addressing, polling for I/O and breakpoints, and if requested, load a file into the System 432/670.

Syntax:

```
DEBUG [filename]
```

Semantics:

The debugger can view System 432/670 memory as: (1) an ordered array of bytes, displayed and modified by physical address; and (2) a logical grouping of segments, displayed and modified by logical address. The debugger can always address memory by means of a physical address, but the DEBUG command must be used before it can recognize logical addresses.

The debugger makes a series of consistency checks on the memory image before logical addressing is enabled. If the contents of memory fail these checks, an error is displayed and logical addressing is not permitted.

The DEBUG command also causes the debugger to begin polling for breakpoints (see Chapter 5) and 432 I/O requests (see Chapter 2, "The Debugger I/O Interface").

If the optional filename is present, the specified file is loaded into 432 memory, as by the LOAD command.

Also see INIT, LOAD, and Chapter 3.

Example:

```
?DEBUG :f2:BUBBLE.EOD
```

enters the file BUBBLE.EOD, located on drive :F2:, into System 432/670 memory, enables logical addressing, and begins polling for breakpoints and I/O.

DIR

List the names in the debugger name table.

Syntax:

```
    ?DIR [type | name]
```

where the possible types are INTEGER, REFERENCE, BREAK, and TEMPLATE.

Semantics:

Used alone, DIR lists all the names in the debugger name table (DNT). If type is used, it lists only names of that type. The debugger provides only a single line of information for each name; for a detailed definition of a particular name, use the command DIR name.

When the DIR command is used to display a single name, the definition displayed is suitable for entering into the debugger. The DIR command can be used together with the LOG command to save definitions of names to a file. This file can be editted (e.g., to remove the prompts and DIR commands) and then used as an INCLUDE file.

Also see REMOVE

Examples

```
    DIR break                          -- display all the breakpoints
    B0          BREAK    BA 9^47.50 of 9^16
    B1         -BREAK    BA 9^49.50 of 9^2b
    B2         -BREAK    BA 9^4B.50 OF 9^40
    B3         *BREAK    BO INST OF 9^16
    B5          BREAK    BO CALL OF 9^16
```

The "-" before BREAK means a process is currently broken at this breakpoint. The "*" means that the breakpoint has been deactivated.

```
    ?DIR context_as           -- display the definition of a template
    TEMPLATE CONTEXT_AS IS
     CTXT_DS:    @0;
     CONST:      @1;
     PREV:       @2;
     MSG:        @3;
     CURR_CTXT: @4;
     EAS_1:      @5;
     EAS_2:      @6;
     EAS_3:      @7;
     DOMAIN:     @8;
    END

    ?DIR cp              --display the current process
    CP IS 9^1B
```

DOWN

Set CC to the next called context in the call stack.

Syntax:

```
    DOWN
```

Semantics:

When the current process (CP) reaches a breakpoint, the debugger sets CC to the address of the breakpointed context. This is considered the bottom of the stack of contexts. CC may be moved up the call stack to its caller using

```
    ?UP
```

and back down to the context which is called using

```
    ?DOWN
```

To find the current value of CC, type:

```
    ?DIR CC
```

If more than one process is at a breakpoint, the user may SELECT a new current process (CP). Before changing to a new CP, the debugger saves the current value of CC: if the current CP is re-selected in the future, CC will be set to this saved value, not the breakpointed context.

Also see TOP, BOTM, SELECT, STACK, and Chapter 5.

EXAMINE

Display the set of breakpointed processes

Syntax:

```
EXAMINE
```

Semantics:

When a process reaches a breakpoint, it is added to the process set; when it is RESUMEd it is removed from the set. The EXAMINE command will display the current contents of the process set. The access descriptor for each process and the value of that process' current context are both displayed.

If the current process is at a breakpoint, then the characters "CP:" will appear to the left of the current process' row of the display (see example).

If a process is de-selected (i.e., another process is SELECTed), the debugger saves the de-selected process' CC value. This is the value diplayed in the EXAMINE command.

Also see BOTM, DOWN, RESUME, SELECT, TOP, UP, and "Processes" in Chapter 5.

Example:

```
?EXAMINE                       -- 4^48 is the current process
      PROCESS         CONTEXT
      4^28            3^23
CP:   4^48            7^5
      4^71            9^13
?DOWN
?EXAMINE                  -- context value for 4^48 should change
     PROCESS         CONTEXT
      4^28            3^23
CP:   4^48            7^1
      4^71            9^13
?SELECT 4^71
?EXAMINE                       -- CC for 4^48 should stay at 7^1
      PROCESS         CONTEXT
      4^28            3^23
      4^48            7^1
CP:   4^71            9^13
?DIR CC
CC IS 9^13
```

EXIT

Close the current log file and leave the debugger.

Syntax:

```
    EXIT
```

Semantics:

At the end of a debugging session, the EXIT command is used to close the log file (if any) and return to the Series III. The EXIT command will appear in the LOG file. The Series III will be in the mode from which the debugger was entered. For instance, if the debugger was invoked from 8080/8085 mode, typing EXIT returns the Series III to that mode. The following message is displayed:

```
    ?EXIT
    -
```

where the "-" prompt indicates 8080/8085 mode. Similarly, if the debugger was invoked from 8086 mode, EXIT returns the Series-III to 8086 mode. From there, the RUN command EXIT may be used to return to 8080/8085 mode.

Also see Chapter 2.

Example:

If you were in 8086 mode before entering the debugger, you will return to 8086 mode when you leave the debugger. From the debugger, type

```
    ?EXIT
    >
```

The ">" prompt indicates 8086 mode. To enter 8085 mode, type:

```
    >EXIT
    -
```

You are now in 8080/8085 mode on the Series III.

INCLUDE

Causes debugger input to be taken from a specified file.

Syntax:

```
    INCLUDE [:fn:]filename [LIST]
```

where n is any valid ISIS-II disk drive number and filename is any valid ISIS-II filename.

Semantics:

INCLUDE causes debugger input to be taken from a specified file until an end-of-file is encountered. After the end-of-file, the input is taken from the previous input source. An INCLUDE file may contain another INCLUDE command; these may be nested up to a depth of four levels.

The console device name, :ci:, may be used as the filename, in which case CONTROL-Z should be used as the end-of-file marker.

If LIST is specified in the command line, the debugger will echo the commands from the include file at the console. Otherwise, the commands are not echoed.

An INCLUDE command may not be followed by another command on the same line. Although the debugger will not indicate an error, it ignores any commands following an INCLUDE.

Input to a 432 process may be INCLUDEd. To do this, the debugger must be in Debugging + I/O mode and each line of input in the INCLUDE file must be preceded by a percent sign (%). For more details see "Entering Input Lines" in Chapter 2

The control characters described in Chapter 2 do not have their usual meanings if they are in an include file.

Example:

A common use of INCLUDE is to enter template definitions for system objects that are usually encountered during debugging. The file DEB432.TEM contains such a series of template definitions, along with commands to set the input and output bases. To make those definitions available during debugging type:

```
    ?INCLUDE :fn:DEB432.TEM
```

The result is that the file DEB432.TEM is used as the debugger input until an end-of-file is encountered. In this way, a number of useful templates are defined and the default number bases are set to 16.

INIT

Initialize the System 432/670.

Syntax:

```
    INIT [SYSTEM addr]
```

where addr is the physical address of the last memory byte in the System 432/670 main memory.

Semantics:

INIT places the System 432/670 in an state capable of executing programs. It resets the 432/670 hardware, clears the 432/670 ECC memory, disables logical addressing, and clears all breakpoints. INIT also determines the address of the last available byte in memory and moves the 256-byte IP control window up to the top of memory. Finally, INIT will display the address of the last byte of memory:

```
    TOP OF MEMORY IS: addr
```

The hardware reset includes resetting all the GDPs and IPs in the System 432/670. This stops all 432 execution.

The INIT SYSTEM command resets the hardware, including the GDPs and IPs in the system, but does not clear the 432/670 memory. Thus, the memory image remains intact and ready for examination.

During either initialization sequence, if the debugger finds that the initialization has failed, it displays the message:

```
    ERR 289: FAILED TO INITIALIZE THE 432 SYSTEM
```

Check that power to the System 432/670 is on and that all connections between the Series III and the 432/670 are in good order. For further information on diagnostic procedures, consult the System 432/600 Diagnostic Software User's Guide. For installation procedures, consule Appendix A of Introduction to the Intel 432 CDS.

Also see DEBUG, LOAD, and Chapter 3.

Examples:

Generally, the INIT and DEBUG commands are used as a pair: INIT to reset the System 432/670 and DEBUG to load memory, enable logical addressing, and start polling for I/O and breakpoints.

```
    ?INIT
    TOP OF MEMORY IS: 7FFFF
    ?DEBUG :f1:bubble.eod
    ?
```

If the debugger's IP "goes fatal," the debugger sends the message:

```
    ERR 329: FATAL IP ERROR
```

The probable causes are:

1. one of the connections between the Series-III and the 432/670 is bad,

2. the software running on the 432/670 system has crashed, or

3. the 432/670 power was turned off.

In the first two cases, the user may want to look at the contents of memory to determine what happened. For this use the INIT SYSTEM command, which preserves the memory image.

When using INIT SYSTEM, you must explicitly specify the address of the last byte in memory as part of the command line, because the debugger cannot determine this address without destroying the current memory image. The debugger uses the last 256 bytes of memory as the control window of the IP. Thus, the last byte of memory available for program execution will be the number you enter minus 256.

IPC

Executes the 432 instructions "Send to Processor" and "Broadcast to Processor"

Syntax:

```
    IPC processor, message
```

where processor is an ordinal expression specifying the processor ID number or the string "ALL". The field message is the code number of the message.

Semantics:

Logical addressing must be enabled for the IPC command to work (see DEBUG). The specified message is either sent to the processor whose ID is specified or, if ALL is used, broadcast to all processors.

The following table gives the IPC encodings and their meanings (see the iAPX 432 GDP Architecture Reference Manual for more information).

0	-	Wakeup
1	-	Start Processor
2	-	Stop Processor
3	-	Set broadcast acceptance mode
4	-	Clear broadcast acceptance mode
5	-	Flush object table cache
6	-	Suspend and fully requalify processor
7	-	Suspend and requalify processor
8	-	Suspend and requalify process
9	-	Suspend and requalify context
10	-	Flush data segment cache
11	-	Enter normal mode
12	-	Enter alarm mode
13	-	Enter reconfiguration mode
14	-	Enter diagnostic mode

Example:

```
    IPC 1,0        -- wake up processor 1

    IPC ALL,2      -- stop all processors

    IPC ALL,1      -- start all processors
```

LOAD

Load a file into System 432/670 memory.

Syntax:

```
    LOAD [:fn:]filename
```

where n is any valid ISIS-II disk drive number and filename is any valid ISIS-II EOD filename.

Semantics:

LOAD is basically a subset of DEBUG. The LOAD command lets you load a 432 EOD file into System 432/670 memory, but does not enable logical addressing.

The LOAD command deactivates all breakpoints (and logical addressing, if it was enabled).

Also see INIT, DEBUG, and Chapter 3.

Example:

To load an object module BUBBLE.EOD, from drive :f2:, type:

```
    ?LOAD :f2:BUBBLE.EOD
```

In general, use the INIT command before a LOAD. Loading a 432 memory image into a system with executing GDPs causes the GDPs to fault, in which case they must be reset with INIT (and memory probably reLOADed) before they can be restarted.

LOG

Record a debugging session.

Syntax:

```
    ?LOG :fn:filename
```

where n is any valid ISIS-II disk drive number and filename is any valid ISIS-II filename.

Semantics:

The LOG command begins a disk file record of all commands entered from the keyboard, as well as debugger output to the console.

When logging is initiated, the first line of the log file is the debugger's sign-on message plus the current date (as most recently set by the RUN command "DATE"). The next line of the log file is the same as the command line that contained the RETURN of the LOG command.

It is possible to direct debugger output to only the log file by typing:

```
    ?>LOG
```

In this case, most debugger output -- with the exception of debugger prompts and error messages -- will not appear on the console. To disable the logging function, and thus cause all debugger input and output to appear at the console only, type the following:

```
    ?>CRT
```

Finally, to return to a mode which directs output to both the log file and the console, type:

```
    ?LOG
```

The log file is closed automatically by the EXIT command.

Do not remove the disk with a log file active until a LOG file has been opened on another drive.

Also see DIR.

Example

```
    ?Log :f1:myprog.log     -- start logging to :f1:myprog.log
    ?1^1:AD
      6^ 17
    ?>log                   -- send console output ONLY to log file
    ?1^1:mem ALL
    ?LOG                    -- and now to both console and log file
    ?
```

MEMORY EXAMINATION

Display bits, bytes, or objects of any form from 432 memory on the CRT.

Syntax:

```
    source [repetition]
or
    source [. field name]...
```

where source is a reference expression, repetition is either LENGTH number or ALL, and field name is the name of a template field.

Semantics:

The address part of the source reference is used to determine the bit stream: which bits from 432 memory are going to be examined.

The template part of the source reference tells the debugger how to display the bit stream at the terminal. If the source reference expression does not have a template part, then the debugger will use the algorithm described in "Default Template Selection Algorithm", in Chapter 4, to select a template.

The field name must be a field of the template used for the source. If present, only that field of the template will be displayed, without the field name. For more on the use of fields see "The Dot Notation" and "The Data Structure Table" in Chapter 4.

Also see "Addresses", "References" and "Examining Memory" in Chapter 4, and Chapter 6 on "Templates".

Examples

```
    ?1^1                  --display processor number 1
    PROCESSOR_AS
    5^ 46   4^ 1B   4^ 45   4^ 33   2^  2   4^ 3F   6^ 21   5^ 43
    0^  0   0^  0   0^  0   0^  0   6^ 19   0^  0   0^  0   5^ 41

    ?4^1B                 --display current process carrier of processor 1
    CARRIER_AS
    4^ 1A   0^  0   0^  0   0^  0   0^  0   4^ 20   4^ 1B   0^  0
    0^  0   4^ 1C   0^  0   0^  0   0^  0   0^  0   0^  0   0^  0
```

```
?cp            -- display the current process access segment
PROCESS_AS
STATUS:          0C04A    [BOUND, FAULTED]
PROC_DS:        4^ 23
CURR_CTXT:      4^ 20
PGLOB_AS:       4^ 1F
LOC_OBJ_TAB:    2^  3
PROC_CARR:      0^  0
DISP_PORT:      4^ 1D
SCHED_PORT:     0^  0
FAULT_PORT:     0^  0
CUR_MSG:        0^  0
CUR_PORT:       0^  0
CUR_CARR:       0^  0
SURR_CAR:       0^  0

?cp:flt        -- display context fault area of the current process.
FAULT_OBJ_IND:    64
PRE_IP:          177      POST_IP:        1B8
PRE_SP:           1C      POST_SP:         1C
FAULT_STATUS:      1
PROC_STATUS:    0C028     PSOR_STATUS:    184
OPERATOR_ID:    10#204#     HISTORY:      ACTIVE
FAULT_CODE:     0FA0E
FAULT_OBJSEL:     8C      FAULT_DISPL:    3E0
```

MEMORY MODIFICATION

Changing some of the values in memory

Syntax:

```
    destination [. field name]... := expression
```

where destination is a reference_expression and field name is the name of a field.

Semantics:

The address part of the destination is used to determine the bit stream: which bits are going to be examined by the template.

The template part of the destination will determine what memory is to be modified, if a field_name is not present. If the template part of the destination is not present, B8 is the default.

If a field name is not present, then the template of the reference is applied to memory as if the memory were going to be examined. The debugger suppresses the display, but keeps track of the high water mark for the template application. All of the memory from the address part of the destination to the high water mark is modified.

If a field name is present, then the memory identified by that field's bit identification component in the template definition (see Chapter 6) is modified.

If the expression on the right hand side of the := is a reference, the default template for that reference is the template used by the right-hand side. The memory is copied from the memory referenced by the right-hand side to the memory referenced by the left-hand side. The memory is copied four bytes at a time, from low address to high address, until the left-hand memory has all been modified. The template on the right-hand side is used to identify what memory will be copied exactly as the template on the left-hand side is used to identify the memory that will be modified. If the amount of memory identified by the right-hand side is not as much as is identified on the left-hand side, the high order bits used to modify the destination will all be zeros.

Also see "Memory Examination", "References", and "Modifying Memory" in Chapter 4.

Examples:

```
    ?cc.eas_1:=cc.domain
```

copy the access descriptor in the domain slot of the current context to entry access list 1.

```
?1^1.2:AD := 0AF01BF          -- 0AF01BF is a 32 bit value for 0A^1B
```

write access descriptor 0A^1B, with full rights, into slot 2 of processor number 1.

```
?3^4'SD:storage.base_addr := 0C1BB0
```

Change the base address field in the Storage descriptor for segment 3^4.

MODE

Display the current debugger I/O mode.

Syntax:

```
    MODE
```

Semantics:

The possible I/O modes are Debugging Only, Debugging + I/O, and I/O Only. These modes are described in "The Debugger I/O Interface", Chapter 2. The control characters used to switch between the modes are:

CONTROL-C	place the debugger in Debugging Only mode
CONTROL-O	place the debugger in I/O Only mode
CONTROL-B	place the debugger in Debugging + I/O mode

The MODE command is used to display the current mode; it cannot be issued from I/O Only mode, since in that mode all input is assumed to be for the System 432/670 execution vehicle.

The debugger will be in Debugging + I/O mode at the start of a session.

Also see "The Debugger I/O Interface" in Chapter 2.

Example:

```
    ?MODE
    DEBUGGING ONLY
    ?
```

OUT

Forget the current access path and clear the DST.

Syntax:

```
    OUT
```

Semantics:

The OUT command tells the debugger to forget about the current access path and to clear out the DST.

Also see BACK, PATH, and the sections on the "Dot Notation" and "Data Structure Table" in Chapter 4.

Example:

```
    ?PATH
    (9^16:context_as).prev.prev.prev(9^0B)
    ?OUT
    ?PATH
    ERR 177: NO PATH TO PRINT:  DST IS EMPTY
```

PATH

Display the current access path.

Syntax:

```
PATH
```

Semantics:

The path command displays the current access path. If there is no current access path, then an error is given.

Only the eight most recent elements of the access path are saved by the debugger. If the current access path is longer than eight elements, then the most recent eight elements are displayed, preceded by three dots (...). This indicates that the earlier portion of the access path has been discarded.

The first element of the access path is always displayed in parentheses in the form of a reference (i.e., address:template). This represents the "anchor" of the displayed path. If the path is preceded by three dots (...) then the anchor is not the first reference keyed-in by the user, but rather the starting point of the visible part of the path.

If the path contains more than one element, it ends with

```
.field_name(di^si)
```

where the field_name is the name of the last field traversed in the access path and the item in parentheses is the value of the access descriptor for that field.

Also see BACK, OUT, and "The Data Structure Table" in Chapter 4.

Example:

```
?9^16:context_as^
?PATH
(9^16:context_as)
?prev^
?PATH
(9^16:context_as).prev(9^12)

...

?PATH
(9^16:context_as).prev.prev.prev.prev(9^7)
?prev.curr_ctxt.curr_ctxt.curr_ctxt.ctxt_ds^
?PATH
...(9^12:context_as).prev.prev.prev.prev.curr_ctxt.curr_ctxt.
curr_ctxt.ctxt_ds(9^1)
```

REMOVE

Remove a name from the DNT.

Sytnax:

```
    REMOVE {name | type | ALL}
```

where name is the symbol to be removed, the possible types are INTEGER, REFERENCE, BREAK, and TEMPLATE.

Semantics:

A name must be in the debugger name table (DNT) to be used in a debugger session. If the user has no more need for a particular name, he can remove it. The debugger will re-use the space occupied by this name for other names. It the DNT becomes too full, it will overflow to disk and accessing names will be slower.

REMOVE name removes a single name from the DNT. REMOVE type removes all names of that type (e.g., all INTEGERs), and REMOVE ALL clears the DNT entirely.

It is not always possible to REMOVE a breakpoint from the DNT, since removing a breakpoint also deactivates the breakpoint; some breakpoints (those that use the 432 tracing mechanism) require that the process be at a breakpoint before the breakpoint can be removed.

Also see DIR.

Examples:

```
    ?REMOVE stkpt          -- remove the name stkpt

    ?REMOVE template       -- remove all templates

    ?REMOVE ALL            -- clear the name table
```

RESTORE

Load a previously SAVEd portion of memory into the System 432/670.

Syntax:

```
    RESTORE [:fn:]filename [TO partition]
```

where n is any valid ISIS-II disk drive number, partition is the area of memory to be loaded, and filename is any valid ISIS-II filename.

Semantics:

RESTORE reloads all or part of a 432 memory image that was stored by a SAVE command.

The partition specification may be entered as:

physical address TO physical address

where the two physical address specifications define the beginning and ending addresses, respectively, of a valid range of 432/670 memory to be reloaded.

Alternatively, partition may define a partition of memory by giving a starting address and length:

physical address LENGTH number

In this case, physical address must be a valid 432/670 memory address, and number must be a non-negative value specifying a partition length within the range of 432/670 memory size.

If the partition specification is omitted, the memory image is copied into the region of memory from which it was previously SAVEd. If the location specified is smaller than the image in the file, the debugger restores only the portion that fits.

The RESTORE command deactivates all breakpoints if the partition includes an instruction segment which has a breakpoint or a process access segment for a process which has an active breakpoint.

Also see SAVE and Chapter 2.

Example:

To reload the image of memory copied to disk by the example SAVE command in the next section, type:

```
    ?RESTORE :f0:TEST.SAV TO !7100 TO !8100
```

The debugger copies the file to the specified memory locations.

RESUME

Continue execution of a breakpointed process.

Syntax:

```
    RESUME [process list | ALL]
```

where process list is a set of logical addresses of process access segments, separated by commas. The default is CP.

Semantics:

RESUME causes a breakpointed process to continue executing user code, starting at the breakpointed instruction.

When CP is resumed, CC is set to the null access descriptor, 0^0.

Also see EXAMINE and SELECT.

Examples:

```
    ?RESUME 4^4      -- resume process 4^4
    ?

    ?RESUME          -- resume the current process
    ?

    ?RESUME ALL      -- resume all processes in the process set
    ?
```

SAVE

Dump all or part of the 432 memory image to an ISIS file.

Syntax:

```
    SAVE partition TO [:fn:]filename
```

where n is any valid ISIS-II disk drive number, partition is the area of memory to be saved, and filename is any valid ISIS-II filename.

Semantics:

The SAVE command lets you dump all or part of a 432/670 memory image to an ISIS-II disk file. The image can later be reloaded with the RESTORE command.

Note that the saved image is NOT in EOD file format.

The partition specification may be specified as:

```
    physical address TO physical address
```

where the two physical address specifications define the beginning and ending addresses, respectively, of a valid range of 432/670 memory to be reloaded.

Alternatively, partition may define a partition of memory by giving a starting address and length:

```
    physical address LENGTH number
```

In this case, physical address must be a valid 432/670 memory address, and number must be an integer specifying a partition length within the range of 432/670 memory size.

Also see RESTORE and Chapter 2.

Examples:

Each of these commands copies the first 10001 bytes of 432/670 memory to a file called TEST.SAV; the first uses starting and ending addresses, and the second uses a starting address and a length.

```
    ?SAVE !0 TO !10000 TO :f0:TEST.SAV

    ?SAVE !0 LENGTH 10001 TO :f0:TEST.SAV
```

'SD ATTRIBUTE

Creates a reference to the object descriptor of the segment of a logical address.

Syntax:

```
reference_expression'SD
```

Semantics:

The 'SD attribute for a reference_expression uses only the address part of a reference. The address must be a logical address.

The attribute is a reference for the object descriptor of the address. The transformation used to go between a logical address and the reference to its object descriptor looks like:

```
i^j.k'SD  =>   2^i.(16*j):DESCR
```

DESCR is supplied as part of DEB432.TEM. It will determine the type of any object table entry and print it in the correct format.

Examples:

```
?cc'SD     -- Display the object descriptor for the current context

?1^1'SD    -- display the object descriptor for processor number 1
           -- (i.e., 2^1.10:descr, assuming base 16)

9^1C'SD:STORAGE.base_addr     -- display base_addr field of
                              -- descriptor (2^9.1C0:descr)
```

In this last example, the template DESCR is overridden with the template STORAGE, that is, 2^9.1C0:descr becomes 2^9.1C0:storage. This enables the dot notation to be used to refer to just the BASE_ADDR field of the object descriptor. The DESCR template supplied in DEB432.TEM has no field names, it uses variants to discover which template should be used to display the object descriptor and then displays the descriptor with that template. STORAGE is the template used to display STORAGE DESCRIPTORS and is required for use of the field names, in this case.

SELECT

Select a process to be the current process.

Syntax:

```
    SELECT process
```

where *process* is the logical address of a process access segment.

Semantics:

The SELECT command changes the value of CP to the specified process, which must be in the set of breakpointed processes, and sets CC to the context value for that process. The new value of CC will be either the breakpointed context, if the process has never been selected since it hit a breakpoint last, or the value of CC when the process was last de-selected.

When a new process is SELECTed, the debugger saves the current value of CC. If the current process is ever re-selected before it is RESUMEd, then this saved value will become the value of CC again.

When the stack of a breakpointed process is displayed, there are two active contexts that are not displayed. These are debugger contexts created to handle the breakpoint.

Also see EXAMINE, RESUME and "Processes", Chapter 5.

Example:

```
    ?dir cp             -- the current process
    ?CP IS 8^18
    ?select 4^14        -- select process 4^14
    ?dir cp             -- new current process
    CP IS 4^14
    ?dir cc
    CC IS 6^1C
    ?up                 -- change cc to 6^1C's caller
    ?dir cc
    CC IS 6^0F
    ?select 8^18        -- de-select 4^14, reselect 8^18
    ?dir cc
    CC IS 0A^7
    ?select 4^14        -- reselect 4^14
    ?dir cc
    CC IS 6^0F
```

This example shows that the debugger "remembers" the last setting of CC for a process.

STACK

Display the call stack of contexts.

Syntax:

```
    STACK [n] [OF process]
```

where n is the number of contexts to display (default is the entire stack), and process is the logical address of any process (the default is CP)

Semantics:

The STACK displays the n most recent contexts on the call stack associated with process. The first line of the display after the column headings is the breakpointed context (i.e., the most recently called context or the BOTM of the stack). Each line that follows is the context that called the context of the previous line. The last line (the TOP of the stack) is the original context in the process, it has no caller.

If process is executing, the results of a STACK command are usually all right, but sometimes unpredictable (for example the process could be executing a return instruction just as the debugger is trying to determine the stack contents).

The INSTRUCTION column shows the instruction segment associated with the context and bit offset where execution will resume in that instruction segment.

If the process is at a breakpoint, then there are two contexts on the stack that are not displayed. These contexts were created to handle the breakpoint and are part of the debugger.

Also see BOTM, DOWN, TOP, and UP.

Examples:

```
    ?STACK
    CONTEXT          INSTRUCTION
    3^10             4^67.112
    3^5              4^66.667
    3^1              4^16.1873
    ?
    ?STACK 1 of 9^16   --display just the location of the breakpoint
    CONTEXT          INSTRUCTION
    10^0C            7^66.50
    ?
```

START

Send the wakeup IPC to a processor

Syntax:

```
START [n]
```

Semantics:

The START command can be used only if logical addressing is enabled (see DEBUG).

With the optional processor number, START is shorthand for:

```
    IPC n, 0.
```

If n is omitted, the debugger searches the processor object table (in the order 1^1, 1^2, 1^3 ...) and wakes up the GDP associated with the first GDP processor object it encounters (i.e., GDP n, where 1^n is the first GDP). The START command is typically used to start a memory image that has just been loaded with a DEBUG, LOAD, or RESTORE command.

Also see IPC and Chapter 2.

SUFFIX

Change the debugger default input base.

Syntax:

```
    SUFFIX [nn]
```

where nn is the desired input base.

Semantics:

The debugger's default base for numeric I/O is 10. The SUFFIX and BASE commands let you set or display the default input and output base.

SUFFIX controls the numeric input base. To set it, use the optional value nn. The value must be a decimal number in the range 2..16 or a number in the form base#number# where base is the desired base in base 10 and number is the desired number in the given base. The decimal equivalent of nn must be in the range 2..16.

All numbers must be entered with a leading digit. If the default input base is greater than 10, precede numbers which begin with A thru F with a zero. For example:

```
    OBAD        -- is ok, whereas
    BAD         -- is assumed to be a name
    16#BAD#     -- is ok, the "1" of 16 is the leading digit
```

To display the current input base, type SUFFIX without a number.

The user can override the default input base when entering any numeric constant by typing:

```
    base#number#
```

where base is a decimal number.

For example if the default base were 10 you could enter a digit in hex by typing:

```
    16#1FFFF#
```

The debugger always displays bases as decimal numbers (between 2 and 16) no matter the current output base.

Also see BASE and Chapter 2.

Examples:

```
    ?SUFFIX 16            -- Change the input base to hexadecimal.
    ?SUFFIX 10            -- Change it back to decimal.
    ?SUFFIX               -- Display the current default input base
    10
```

TOP

Set CC to the initial context of CP.

Syntax:

```
    TOP
```

Semantics:

When the current process reaches a breakpoint, CC is assigned the value of the breakpointed context, considered the bottom of the call stack. To set CC to the initial context of CP, type:

```
    ?TOP
    ?
```

If more than one process is at a breakpoint, the user may SELECT a new current process (CP). Before changing to a new CP, the debugger saves the current value of CC: if the current CP is re-selected in the future, CC will be set to this saved value, not the breakpointed context.

The inverse command is BOTM. Also see UP, DOWN, STACK, and Chapter 5.

UP

Set CC to the context of the caller of the current context.

Syntax:

```
    UP
```

Semantics:

When the current process reaches a breakpoint, CC is assigned the value of the breakpointed context, considered the bottom of the call stack. To move the value of CC to its caller (the previous context field in the context access segment), type:

```
    ?UP
    ?
```

If more than one process is at a breakpoint, the user may SELECT a new current process (CP). Before changing to a new CP, the debugger saves the current value of CC: if the current CP is re-selected in the future, CC will be set to this saved value, not the breakpointed context.

The inverse command is DOWN. Also see SELECT, TOP, BOTM, STACK, and Chapter 5.

VERSION

Display the version number of the current DEBUG-432 debugger.

Syntax:

```
    VERSION
```

Semantics:

It is useful to know which version of the debugger you are running; different versions may have different features. The VERSION command provides an easy method of identifying the version of the debugger without having to exit and subsequently reenter the debugger.

The debugger indicates the current version and also displays the address of the highest available memory location, the current I/O mode of the debugger, and the status of any logging activity (including the log filename, if any). The format of the reply is:

```
    ?VERSION
    SERIES III  432 SYSTEMS LEVEL DEBUGGER, Vx.yz
    TOM:  !yyyyy    MODE: mode    LOG FILE: :fn:filename  {ON|OFF}
```

where x.yz is the current version number, TOM stands for Top Of Memory, !yyyy is the address of the last byte in memory, mode is one of the three I/O modes of the debugger (Debugging Only, Debugging + I/O, or I/O Only), the log file is specified as an ISIS-III filename, and the logging facility is either ON or OFF.

Also see INIT, LOG, MODE, and "The Debugger I/O interface", Chapter 2.

Example:

```
    ?VERSION
    SERIES III  432 SYSTEMS LEVEL DEBUGGER, V1.00
    TOM: !7FFFF    MODE: DEBUGGING ONLY     LOG FILE: :F1:MYLOG (ON)
```

PART II

UPDATE - 432

INTRODUCTION

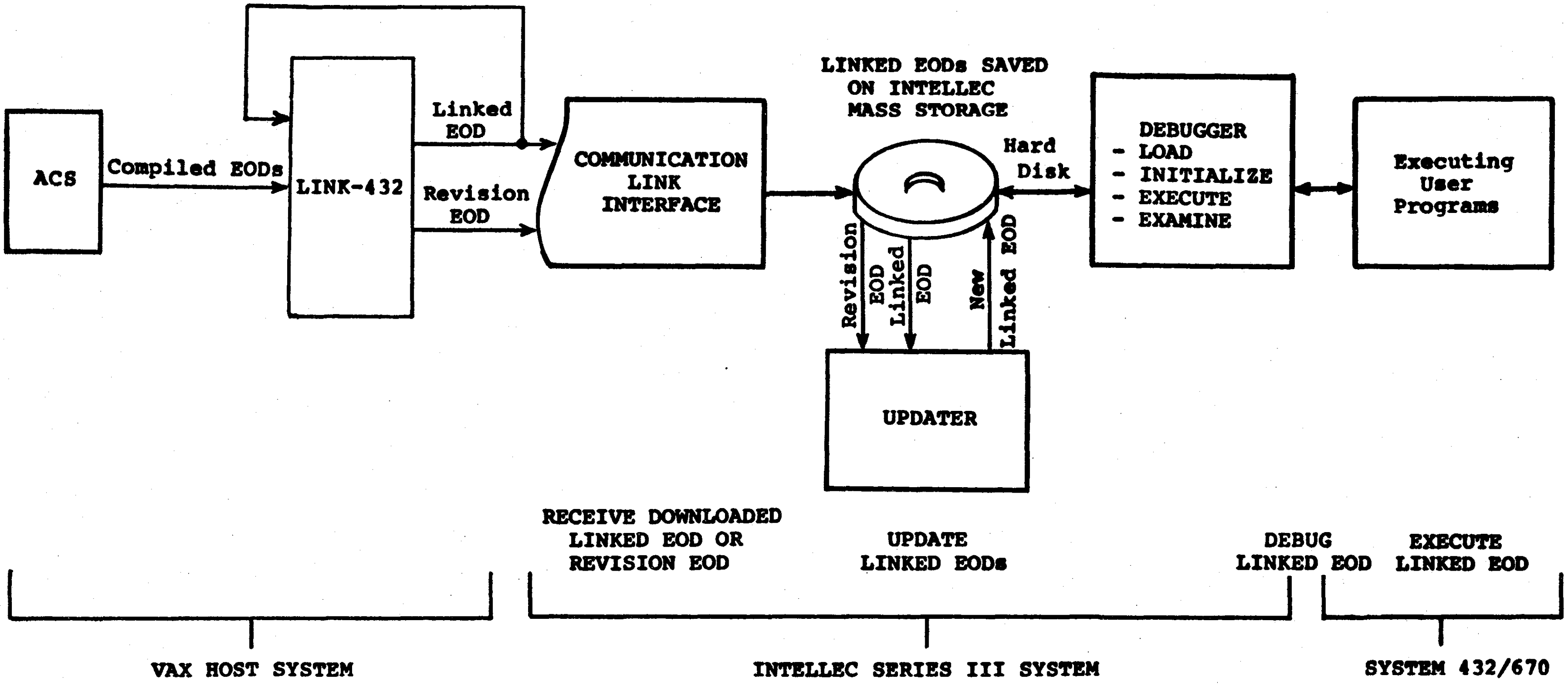
The purpose of UPDATE-432 is to let the user quickly revise a linked External Object Description (EOD) on the Series III system, when the corresponding Ada source program has been modified on the mainframe host. UPDATE-432 accomplishes this revision by operating on the linked EOD using information contained in a revision EOD. The revision EOD is produced by LINK-432 when the newly compiled Ada program is relinked, and is sent over the communication link to the Series III.

REVIEW OF COMPILING, LINKING, AND EODs

The Ada Compiler System supports separate compilation, the division of a program into modules that can be designed, developed, and compiled separately. The LINK-432 linker combines these separately compiled modules into a single file. It also assigns physical addresses to 432 segments, object table indices (segment coordinates), and base addresses in object descriptors.

The linker outputs a linked EOD file that contains the modules linked together. If the linker input included a linked EOD, the output includes a revision EOD file containing the changes made to the input linked EOD to produce the output linked EOD. Revision EODs contain new modules, the names of any modules to be deleted, and address changes. Since the revision EOD records only the differences between the input and output linked EOD files, it can be significantly smaller than a linked EOD file. (Consult the Intel 432 CDS VAX/VMS Host User's Guide for a full description of how to generate a revision EOD.)

Since the bandwidth of the communication link between the mainframe and the Series III is relatively limited, it is desirable to reduce the amount of redundant information that must be downloaded. Using UPDATE-432, it is not necessary to download the entire newly linked EOD every time the program is changed. Instead, only the revision EOD need be sent to the Series III. UPDATE-432 uses the information in this revision file to update the linked EOD on the Series III. Figure 8-1 shows how the programs in the 432 CDS interact through EOD files.



F-0259

Figure 8-1. Updating Linked EOD Files

PRINCIPLE FUNCTIONS OF UPDATE-432

UPDATE-432 peforms three functions:

- EOD compatibility check
- EOD module updating
- segment address and object descriptor updating

VERIFYING EOD COMPATIBILITY

To update a linked EOD, UPDATE-432 must verify that the revision EOD sent over the communciation link is matched correctly with the linked EOD to be updated. All EODs, including revision EODs, contain a unique identifier. A revision EOD also contains an identifier that specifies which linked EOD it revises. UPDATE-432 checks these unique identifiers of the revision and linked EODs. If they match, the revision is performed; if they do not match, UPDATE-432 informs the user with an error message and returns to ISIS.

EOD MODULE UPDATING

Updating the modules in a linked EOD is performed by two basic functions: module insertion and module deletion.

Module Insertion: New modules generated by the linker and are passed to UPDATE-432 in the revision EOD. UPDATE-432 adds the object description representing each new module to the existing linked EOD to produce a new linked EOD.

Module Deletion: Any modules to be deleted are named in a delete list created by the linker and passed to UPDATE-432 in the revision EOD. UPDATE-432 deletes the object description representing each module from the linked EOD, producing a new linked EOD.

SEGMENT ADDRESS AND OBJECT DESCRIPTOR UPDATING

One of the essential functions of UPDATE-432 is reassigning the physical addresses of 432 segment images contained in the linked EOD. The base addresses of 432 segments and base addresses contained in 432 object descriptors generally change each time a program is processed by LINK-432. This change occurs because LINK-432 relocates program segments as it links them. The old base addresses and base addresses in object descriptors must be revised each time a linked EOD is updated.

A list of physical base addresses of segments and addressing coordinates is contained in the revision EOD. UPDATE-432 uses this address list to reassign the new physical base addresses of segments and update the base addresses contained in object descriptors.

CHAPTER 9
USING UPDATE-432

INTRODUCTION

Like DEBUG-432, UPDATE-432 is an 8086 program that executes under the control of RUN 8086. The main difference in invoking UPDATE-432 is that it accepts parameters in the command line that invokes it.

To use UPDATE-432, the user invokes it through the RUN program, supplying one required parameter and up to two optional parameters in the command line. The required parameter is the name of the linked EOD to be updated. The two optional parameters specify non-default names for the revision EOD and the newly-updated linked EOD. This section shows how to invoke UPDATE-432.

The Series III has two execution modes: the 8080/8085 mode under control of ISIS-II, and the 8086 mode under control of RUN 8086. When the Series III system starts running, it is in 8080/8085 mode. You can invoke UPDATE-432 directly from 8080/8085 mode by using the RUN command, which has the format:

```
    [:fm:]RUN [:fn:]filename [parameters] [;comments]<RETURN>
```

where

:fm: refers to the disk drive that contains the RUN program. The value m is an integer between 0 and 9. If :fm: is not specified, :f0: is assumed.

filename is the name of the 8086 program (in this case, UPDATE). Again, :fn: refers to a valid ISIS disk drive, this time the drive that contains filename.

parameters are one or more data items required by the 8086 program (in this case, the names of the EODs).

comments are one or more ASCII characters, not including carriage return or line feed. Comments always begin with a semicolon.

For example,

```
    -RUN MYPROG PARAM1 PARAM2 PARAM3; this is a comment<RETURN>
```

For more information on the RUN command and RUN 8086 program, see the Intellec Series III Microcomputer Development System Console Operating Instructions.

Line Continuation:

When a RUN command requires more than one line, terminate each intermediate line with an ampersand (&) followed by a carriage return. A comment may be inserted between the ampersand and the carriage return. Up to 120 characters may be entered before the ampersand.

When the RUN command is ready to accept a continued command line, it prompts with two angle brackets (>>).

For example,

```
-RUN MYPROG PARAM1 & this is a comment<RETURN>
>>PARAM2 & this is also a comment<RETURN>
>>PARAM3; this is the last comment<RETURN>
```

THE UPDATE-432 COMMAND LINE

When using RUN to execute UPDATE-432, the filename is UPDATE. The parameters field contains three parameters, the first is required and the other two are optional. The required parameter is simply the name of the linked EOD to be revised. The two optional parameters are supplied by including directives in the command line. The UPDATE command line thus has the following format:

```
[:fm:]RUN  [:fn:]UPDATE  [:fk:]linked EOD [directives]<RETURN>
```

where square brackets indicate optional items. If a linked EOD is not specified, UPDATE-432 signals a fatal error and returns to ISIS. Directives are discussed in the following sections.

UPDATE-432 DIRECTIVES

UPDATE-432 recognizes two directives, REVISION and NEW. The REVISION directive specifies a non-default revision EOD filename; the NEW directive specifies a non-default linked EOD filename. These directives may be omitted entirely, either may be used by itself, or they can be used together.

THE DEFAULT CASE (NO DIRECTIVES)

The default case for invoking UPDATE-432 is with no directives. Only the linked EOD is specified. In this case, the revision EOD is assumed to have the same filename as the linked EOD, except the extension is assumed to be .REV. During execution, the updater creates a temporary file, :fn:UPD001.TMP, to hold the new linked EOD; where n is the drive containing the old linked EOD. When the operation is complete, the old linked EOD file is deleted and the temporary file is given the same name as the old file. (This temporary file is always created if the NEW directive is not specified.) If the default revision EOD does not match the specified linked EOD, UPDATE-432 prints an error message and returns to ISIS.

For example, (we assume that all files are on :f0:):

```
    -RUN UPDATE PROG.LNK<RETURN>
```

In this example, the linked EOD to be updated is assigned the name PROG.LNK; the revision EOD is assigned the name PROG.REV; and the newly revised linked EOD is also assigned the name PROG.LNK. The old linked EOD is replaced.

THE REVISION DIRECTIVE ALONE

The user may specify a non-default filename for the revision EOD by using the REVISION directive. The format of this directive is

```
REVISION(:fn:filename )
```

where n is the disk drive number and filename is the name of the revision EOD file. Thus, the complete UPDATE command line format using this directive is as follows (assuming :f0: in each case):

```
    RUN UPDATE linked EOD REVISION( revision EOD )<RETURN>
or
   RUN UPDATE linked EOD RE( revision EOD )<RETURN>
```

For example

```
    -RUN UPDATE PROG.LNK REVISION( RPROG.REV )<RETURN>
```

In this example, the linked EOD to be updated is assigned the name PROG.LNK; the revision EOD is assigned the name RPROG.REV; and the newly-revised linked EOD is also assigned the name PROG.LNK. The old linked EOD is replaced. Again, if the revision EOD does not match the linked EOD, UPDATE-432 prints an error message and returns to ISIS. If the updater runs out of memory while processing a revision file, it creates two temporary files, :fn:UPD002 and :fn:UPD003, to hold the overflow; where n is the drive containing the revision file. (Such an occurence is extremely unlikely, since more than 7,000 updated objects must be specified in the revision file.)

THE NEW DIRECTIVE ALONE

The user may specify a non-default filename for the newly-updated linked EOD by using the NEW directive. The format of this directive is

```
NEW(:fn:filename )
```

where n is the disk drive number and filename is the name of the newly-revised linked EOD file. NEW may be abbreviated as NE, if so desired. Thus, the complete UPDATE command line format using this directive is as follows (assuming :f0:)

```
    RUN UPDATE linked EOD NEW( new linked EOD )<RETURN>
```

For example

```
    -RUN UPDATE PROG.LNK NEW( PROG1.LNK )<RETURN>
```

In this example, the linked EOD to be updated is assigned the name PROG.LNK; the revision EOD is assigned the default name PROG.REV; and the newly-revised linked EOD is assigned the name PROG1.LNK. The old linked EOD is not overwritten. If the revision EOD and linked EOD do not match, UPDATE-432 sends and error message and returns to ISIS.

USING BOTH DIRECTIVES

The user may specify non-default filenames for both the revision EOD and the newly-updated linked EOD by using both directives in the same command line. The complete UPDATE command format using both directives is as follows (again assuming :f0: in each case and also using multiple lines):

```
    RUN UPDATE linked EOD REVISION(revision EOD ) &
    NEW( new linked EOD ) <RETURN>
```

For example

```
    -RUN UPDATE PROG.LNK REVISION( RNAME.REV )
    >>NEW( NEWPROG.LNK )<RETURN>
```

In this example, the linked EOD to be updated is assigned the name PROG.LNK; the revision EOD is assigned the name RNAME.REV; and the newly-revised linked EOD is assigned the name NEWPROG.LNK. The old linked EOD is not overwritten. If the linked EOD and revision EOD do not match, UPDATE-432 sends an error message and returns to ISIS.

APPENDIX A
FORMAL DEFINITION OF DEBUG-432 COMMAND SYNTAX

This appendix contains a formal definition of the DEBUG-432 command syntax. The definition uses a variant of Backus-Naur Form (BNF). The following conventions are used:

<identifier>	An identifier in angle brackets is expanded in another line. E.g. <template_definition>
upper_case_ids	Keywords are in upper case. E.g. DEBUG, TEMPLATE, ALL
lower_case_ids	Lower case identifiers denote lexical classes E.g. identifier, file_name
"abc"	Character strings in double quotes stand for literal items. E.g. ")", "=>"
[...]	Square brackets enclose optional items.
(...)	Parenthesis enclose several items; one of these items must be used.
{ ... }	Braces surround an item or set of items which may be repeated zero or more times.
--	A double hyphen precedes comments
a ¦ b	A vertical line denotes exclusive or.
!	Concatenate the characters on either side of the exclamation point.

The lexical classes are:

```
identifier     Identifiers are as in Ada:
               identifier      ::= letter {["_"] letter_or_digit}
               letter or digit ::= letter ¦ digit
               letter ::= upper_case_letter ¦ lower_case_letter
               E.g. process_8_16, CURRENT_CONTEXT

ordinal        Ordinals are as follows:
               digit          ::= digit {["_"] extended_digit}
               extended_digit ::= digit ¦ A ¦...¦ F ¦ a ¦...¦ f
               E.g. 2048, 0A12
```

```
    based_ordinal  Based_ordinals are similar to based numbers in Ada:
                   based_ordinal  ::= base "#" based_integer "#"
                   base           ::= digit [digit]
                   based_integer  ::= extended_digit {["_"]
                                                extended_digit}
                   E.g. 16#ffff#, 2#1010#
```

string — A string is zero or more Ada characters between double-quotes. A double quote within the string is represented by two double-quotes.
E.g. "system rights", "enter ""YES"" or ""NO"" "

file_name — A valid ISIS-II/86 filename.

COMMAND SYNTAX

```
<command_list>  ::=   [<command_list> ";"]  <command>


<command> ::=
                <system_control_commands>
             |  <environment_control_commands>
             |  <memory_contents_filing>

             |  <breakpoint_commands>
             |  <broken_process_commands>
             |  <call_stack_commands>

             |  <template_definition>
             |  <reference_definition>
             |  <integer_definition>
             |  <directory_commands>

             |  <memory_examination>
             |  <name_scope_commands>
             |  <memory_modification>
```

SYSTEM CONTROL COMMANDS

```
system_control_commands> ::=
                INIT [(SYS | SYSTEM) <physical_address>]
             |  LOAD file_name
             |  DEBUG [file_name]
             |  START [<number>]
             |  IPC (<expression> | ALL) "," <expression>
```

ENVIRONMENT CONTROL COMMANDS

```
<environment_control_commands> ::=
              EXIT
           |  INCLUDE file_name [LIST]
           |  BASE    [<number>]
           |  SUFFIX  [<number>]
           |  LOG [file_name]
           |  ">" CRT
           |  ">" LOG
           |  MODE
           |  VERSION
```

MEMORY CONTENTS FILING

```
<memory_contents_filing> ::=
              SAVE <partition> TO file_name
           |  RESTORE file_name [TO <partition>]

<partition> ::=
              <physical_address> TO <physical_address>
           |  <physical_address> (LEN | LENGTH) <number>
```

BREAKPOINT COMMANDS

```
<breakpoint_commands> ::=
              [<break_id>] <brk> [<break_processes>]
           |  ACTIVATE        (identifier | ALL)
           |  DEACTIVATE      (identifier | ALL)

<break_id> ::=        identifier ":"

<brk> ::=
              <break_at>
           |  <break_enter>
           |  <break_exit>
           |  <break_on>

<break_at> ::=        BA <primary>

<break_enter> ::=     BE <primary>

<break_exit> ::=      BX <primary>

<break_on> ::=        BO (INST | CALL | RET | FAULT)

<break_processes> ::= OF (<process_list> | ALL)

<process_list> ::=    [<process_list> ","] <primary>
```

BROKEN PROCESS COMMANDS

```
<broken_process_commands> ::=
              EXAMINE
           ¦  RESUME [<process_list> ¦ ALL]
           ¦  SELECT <expression>
```

CALL STACK COMMANDS

```
<call_stack_commands> ::=
              TOP
           ¦  BOTM
           ¦  UP
           ¦  DOWN
           ¦  STACK [<number>] [OF <primary>]
```

TEMPLATE DEFINITION SYNTAX

```
<template_definition> ::=
              TEMPLATE identifier IS
                      <component_list>
              END

<component_list> ::=  <field_list> [<variant_part>]

<field_list> ::=      [<field_list> ";"] <field>

<field> ::=
             [<field_ident>] [<bit_identification>] [IS <display_list>]

<field_ident> ::=
              identifier ":"
           ¦  identifier "::"

<bit_identification> ::=
              <bit_descriptor>
              "@" <number> "." <bit_identification>

<bit_descriptor> ::=
              "@" <number> [ACCESS identifier]
           ¦  <bit_string>
           ¦  identifier

<bit_string> ::=
              "[" <byte_start> <bit_start> "," <bit_length> "]"

<byte_start> ::=      <template_expr>

<bit_start> ::=       [":" <template_expr>]

<bit_length> ::=      <template_expr>
```

```
<template_expr> ::=
              <template_expr> "+" <template_term>
           ¦  <template_expr> "-" <template_term>
           ¦  <template_term>

<template_term> ::=
              <template_term> "*" <template_prim>
           ¦  <template_term> "/" <template_prim>
           ¦  <template_prim>

<template_prim> ::=
              <number>
           ¦  identifier
           ¦  "(" <template_expr> ")"

<display_list> ::=    [<display_list> ","] [<repetition>] <disp_elem>

<repetition> ::=      "<" template_expr ">"

<disp_elem> ::=

              "[" <display_list> "]"
           ¦  <integer> [":" <width>]
           ¦  "(" <enum_list> ")"
           ¦  string
           ¦  ordinal!"X"
           ¦  ASCII
           ¦  "/"
           ¦  identifier

<integer> ::=
              ordinal!"U"
            ¦ ordinal!"S"

<width> ::=           ordinal

<enum_list> ::=       [<enum_list> ","] <enum_item>

<enum_item> ::=       [case_number "=>"] <enum_value>

<enum_value> ::=
              identifier
           ¦  string

<variant_part> ::=
              CASE <bit_identification> IS
                      [<case_list>]
              END CASE

<case_list> ::=       [case_list] <single_case>

<single_case> ::=     WHEN <choice_list> "=>" <component_list>

<choice_list> ::=     [<choice_list> "¦"] <choice>
```

```
<choice> ::=
              <case_number>
           ¦  <case_range>
           ¦  OTHERS

<case_range> ::=      <case_number> ".." <case_number>

<case_number> ::=
              "-" <number>
           ¦  <number>
```

REFERENCE DEFINITION

```
<reference_definition> ::=
              identifier IS (<template_appl> ¦ identifier)
```

INTEGER DEFINITION

```
<integer_definition> ::=
                      identifier ":" INTEGER [":=" <expression>]
```

DIRECTORY COMMAND SYNTAX

```
<directory_commands> ::=
              REMOVE (identifier ¦ <type> ¦ ALL)
           ¦  DIR [identifier ¦ <type>]

<type> ::=
              INTEGER
           ¦  TEMPLATE
           ¦  REFERENCE
           ¦  BREAK
```

MEMORY EXAMINATION SYNTAX

```
<memory_examination> ::=
              <expression> [<disp_len>]
           ¦  "."
           ¦  <template_application> "." [ALL]
           ¦  identifier "." [ALL]

<disp_len> ::=
              (LEN ¦ LENGTH) <number>
           ¦  ALL
```

NAME SCOPE COMMANDS

```
<name_scope_commands> ::=
                identifier "^"
             |  <template_application> "^"
             |  BACK
             |  PATH
             |  OUT
```

MEMORY MODIFICATION SYNTAX

```
<memory_modification> ::=
                   (identifier | <template_application>) ":=" <expression>
```

EXPRESSION SYNTAX

```
<expression> ::=
                <expression> "+" <signed_term>
             |  <expression> "-" <signed_term>
             |  <signed_term>

<signed_term> ::=
                "+" <term>
             |  "-" <term>
             |  <term>

<term> ::=
                <term> "*" <factor>
             |  <term> "/" <factor>
             |  <term> "REM" <factor>
             |  <term> "MOD" <factor>
             |  <factor>

<factor> ::=
                <factor> "**" <primary>
             |  <primary>

<primary> ::=
                <template_application>
             |  <paren_expr>

<template_application> ::=
                <template_application> <template_options>
             |  identifier <template_options>
             |  <address>
             |  "." identifier

<template_options> ::=
                "." <restricted_expr>
             |  "." identifier
             |  ":" identifier
             |  "'" SD
             |  "!" <paren_expr>
```

ADDRESS SYNTAX

```
<address> ::=
                <logical_address>
             |  <physical_address>
             |  <interconnect_address>

<logical_address> ::=       <restricted_expr> "^" <restricted_expr>

<physical_address> ::=      "!" <paren_expr>

<interconnect_address> ::=  "!" "!" <paren_expr>

<paren_expr> ::=
                <restricted_expr>
             |  identifier

<resticted_expr> ::=
                <number>
             |  "(" <expression> ")"
<number> ::=
                ordinal
             |  based_ordinal
```

APPENDIX B
DEBUG-432 RESERVED WORDS AND COMMAND SUMMARY

This appendix contains a list of all DEBUG-432 reserved words, as well as a summary of debugger commands.

RESERVED WORDS

The following keywords are reserved by DEBUG-432 and must not be used for user-defined symbols:

ACCESS	ACTIVATE	ALL	ASCII
BA	BACK	BASE	BE
BO	BOTM	BREAK	BX
CALL	CASE	CRT	DEACTIVATE
DEBUG	DIR	DOWN	END
EXAMINE	EXIT	FAULT	INCLUDE
INIT	INST	INTEGER	IPC
IS	LEN	LENGTH	LIST
LOAD	LOG	MOD	MODE
OF	OTHERS	OUT	PATH
REFERENCE	REM	REMOVE	RESTORE
RESUME	RET	SAVE	SD
SELECT	STACK	START	SUFFIX
SYS	SYSTEM	TEMPLATE	TO
TOP	UP	VERSION	WHEN

SPECIAL NAMES

In addition to the reserved words listed above, the debugger has special uses for some unreserved standard names. In particular, the debugger assigns special meanings to the four names:

CC   CP   By_P   Bi_P

and to the template names listed in Tables 4-2 and 4-3. Although the user can redefine these special debugger names, it is not advisable to do so.

COMMAND SUMMARY

The following list contains the set of DEBUG-432 commands with a short description of each.

Command	Function
ACTIVATE	Return a breakpoint to the set of enabled breakpoints.
BA	Set a breakpoint at an instruction in an instruction segment.
BACK	Go back one element in the access path.
BASE	Display or alter the debugger's output base.
BE	Breakpoint on entry to a specified instruction segment.
BO	Breakpoint after every specified event.
BOTM	Change the value of CC to the address of the last called context on the stack.
BX	Breakpoint before exit from specified instruction segment.
>CRT	Direct all debugger dialog to the console only.
DEACTIVATE	Remove a breakpoint from the set of enabled breakpoints.
DEBUG	Enable logical addressing and begin polling for breakpoints and I/O.
DIR	Display all or selected parts of the debugger name table.
DOWN	Set the value of CC to the next called context in the call stack.
EXAMINE	Display the contents of the breakpointed process set.
EXIT	Leave the debugger and return to the ISIS-II mode that invoked the debugger.
INCLUDE	Take the debugger command stream from the specified file until an end-of-file is encountered.
INIT	Reset the 432/670 hardware and memory.

INIT SYS — Reset the 432/670 hardware, leaving memory intact.

IPC — Send the specified interprocessor communication message to the specified processor.

LOAD — Copy the contents of the specified 432 EOD file into the 432/670 memory.

LOG — Echo the debugger console dialogue in the specified file.

MODE — Display the current debugger I/O mode of operation.

OUT — Clear the current access path.

PATH — Display the current access path

REMOVE — Delete the specified name or type from the debugger name table.

RESTORE — Load the contents of a SAVEd file into the System 432/670 memory.

RESUME — Send the specified process(es) back to a GDP for execution.

SAVE — Copy the specified 432/670 memory locations to the specified file.

SELECT — Select the specified process to be the current process.

STACK — Display the context stack for a process.

START — Send an IPC wakeup message to the lowest-numbered GDP in the 432/670 system.

SUFFIX — Display or alter the debugger's input base.

TEMPLATE — Define a debugger template.

TOP — Set the value of CC to the top of the call stack (oldest context).

UP — Change the value of CC to the next higher position in the call stack (i.e. to its caller).

VERSION — Display the debugger version number, I/O mode, logging status, and top of memory.

APPENDIX C
DEBUG-432 TEMPLATES IN DEB432.TEM

This appendix contains a list of the presupplied templates and their definitions.

PRESUPPLIED TEMPLATES

- ad
- a1
- as
- b8
- b16
- b32
- bs
- ch
- context_as
- context_ds
- descr
- ds
- dump
- extract
- f_area
- f_or_t
- flt
- free
- header
- interconnect
- mem
- pflt
- proc_stat
- process_as
- psorflt
- rad
- ras
- refine
- so
- storage
- system_type
- type_des

TEMPLATE DEFINITIONS

```
    TEMPLATE ad IS
         [2:4,12] IS Ou:3, "^";
         [0:4,12] IS Ou:3;
    END


    TEMPLATE as IS
         [by_p:bi_p+1,32] IS <2>[<8>[ad, 2x],/];
    END


    TEMPLATE b8 IS    --  8 bits
         [0,8] IS Ou;
    END


    TEMPLATE b16 IS
         [0,16] IS Ou;
    END


    TEMPLATE b32 IS
         [0,32] IS Ou;
    END


    TEMPLATE bs IS
         [0,32] IS Ou;
    END


    TEMPLATE ch IS
         CASE [0,8] IS
           WHEN 16#20#..16#7d# => [0,8] IS ascii
           WHEN others => IS ".";
         END CASE
    END

    TEMPLATE context_as IS
         ctxt_ds:         @0;
         const_ds:        @1;
         prev:            @2;
         msg:             @3;
         curr_ctxt:       @4;
         eas_1:           @5;
         eas_2:           @6;
         eas_3:           @7;
         domain:          @8;
    END
```

```
TEMPLATE context_ds IS
     ctxt_status:    [0,1]   IS "[",(not_faulted,faulted);
                     [0,16]  IS ", value: ", Ou, "]",/;
     sp:             [2,16]  IS Ou:5,/;
     inst_idx:       [4,16]  IS Ou:5,3x;
     ip:             [6,16]  IS Ou:5,/;
     trace_idx:      [8,16]  IS Ou:5,3x;
     trace_ip:       [10,16] IS Ou:5,/;
     trace_code:     [12,16] IS Ou:5,/;
     [by_p:bi_p+1,16]  IS    "OE..OF ",   Ou:9,/,
                             "10..1F ",<8>Ou:9,/,
                             "20..2F ",<8>Ou:9,/,
                             "30..3F ",<8>Ou:9;
END

TEMPLATE descr IS
CASE [0,2] IS
     WHEN 0 =>
       CASE [0:3,2] IS
           WHEN 0 =>
             CASE [0:2,1] IS
                 WHEN 0 =>
                     [0, 128] IS header;
                 WHEN 1 =>
                     [0, 128] IS free;
             END CASE
           WHEN 1 =>
             [0, 128] IS interconnect;
       END CASE
     WHEN 1 =>
       [0, 128] IS type_des;
     WHEN 2 =>
       [0, 128] IS refine;
     WHEN 3 =>
       [0, 128] IS storage;
END CASE
END

TEMPLATE ds IS
     [by_p:bi_p+1,16] IS <4>[<8>Ou:9,/];
END


TEMPLATE dump IS
     [by_p:bi_p+1,8] IS <16>Ou:3, 3x;
     [0,8] IS "'",ch;
     [by_p:bi_p+1,8] IS <15>ch, "'";
END


TEMPLATE extract IS
     [0, 16] IS Ou
END
```

```
TEMPLATE f_area IS
     fault_objind:   [0, 16] IS 16u:5, /;
     pre_ip:         [4, 16] IS 16u:5, 3x;
     post_ip:        [2, 16] IS 16u:5, /;
     pre_sp:         [8, 16] IS 16u:5, 3x;
     post_sp:        [6, 16] IS 16u:5, /;
     fault_status:   [10, 16] IS 16u:5, /;
     proc_status:    [12, 16] IS 16u:5, 3x;
     psor_status:    [14, 16] IS 16u:5, /;
     operator_id:    [16, 16] IS 10u:5, /;
     history:        [16, 15,1] IS (active,handled),/;
     fault_code:     [18, 16] IS 16u:5, /;
     fault_objsel:   [20, 16] IS 16u:5, 3x;
     fault_displ:    [22, 16] IS 16u:5, /;
END

TEMPLATE f_or_t IS
     [0,1] IS (f,t);
END


TEMPLATE flt IS
     @0.[64, 16#0ffff#] IS f_area;
END


  TEMPLATE free IS
     descr_type:     [0,1] IS "free_descr",/;
     free_index:     [2:4,12];
END


TEMPLATE header IS
     descr_type:     [0,1] IS "header",/;
     free_index:     [2:4,12];
     end_index:      [4:4,12];
     fault_level:    [6,16];
     level_no:       [10,16];
     claim:          [12,32] IS 0u;
END

TEMPLATE interconnect IS
     descr_type: IS "interconnect",/;
     valid:          [0:2,1] IS f_or_t,/;
     windowed:       [0:5,1] IS f_or_t,/;
     base_addr:      [1,24];
     length_:        [4,16];
     copied:         [9:1,0];
     level:          [10,16];
END

TEMPLATE mem IS
     [by_p:bi_p+1,128] IS <4>[dump,/];
END
```

```
TEMPLATE ord IS
     [0,32] IS 0u;
END


TEMPLATE pflt IS
     @0.[112, 16#0ffff#] IS f_area;
END


TEMPLATE proc_stat is
     [0,16] IS 0u,3x;
     [0,1]  IS "[", (bound, not_bound),2x;
     [0:1,1] IS (not_faulted, faulted), "]";
END

  TEMPLATE process_as IS
     status:         @0.[36,16] IS proc_status,/;
     proc_ds:        @0;
     curr_ctxt:      @1;
     pglob_as:       @2;
     loc_obj_tab:    @3;
     proc_carr:      @4;
     disp_port:      @5;
     sched_port:     @6;
     fault_port:     @7;
     cur_msg:        @8;
     cur_port:       @9;
     cur_carr:       @10;
     surr_carr:      @11;
END


TEMPLATE psorflt IS
     @0.[16, 16#0ffff#] IS f_area;
END


TEMPLATE rad IS          --"rad" stands for Raw Access Descriptor
     [2:4,12] IS 0u, "^";
     [0:4,12] IS 0u;
     [2:3,1]  IS " wrhd: ",f_or_t;   -- write rights
     [2:2,1]  IS f_or_t;             -- read rights
     [2:1,1]  IS f_or_t;             -- unchecked copy rights
     [2,1]    IS f_or_t;             -- delete rights
     [0:1,3]  IS " type: ",2u;        -- system rights
     [0,1]    IS " valid: ",f_or_t;
END


TEMPLATE ras IS
     [by_p:bi_p+1,32] IS <14>[rad,/],rad;
END
```

```
    TEMPLATE refine IS
         descr_type:     [0,1]   IS "refinement",/;
         valid:          [0:2,1] IS f_or_t,/;
         base_type:      [0:3,1] IS (DS,AS),/;
         sys_type:       [0,128] IS system_type, /;
         bypass:         [0,32]  IS ad,/;
         base_displ:     [6,16];
         length_:        [4,16];
         psor_class:     [8:5,3];
         copied:         [9,f_or_t];
         level_no:       [10,16];
         source_ad:      [12,rad];
    END

    TEMPLATE so IS
         [0,16] IS 0u;
    END

    TEMPLATE storage IS
         type:           [0,2] IS "storage",/;
         valid:          [0:2,f_or_t];
         base_type:      [0:3,1] IS (DS,AS),/;
         sys_type:       [0,128] IS system_type,/;
         allocated:      [0:4,f_or_t];
         windowed:       [0:5,f_or_t];
         altered:        [0:6,f_or_t];
         accessed:       [0:7,f_or_t];
         base_addr:      [1,24];
         length_:        [4,16];
         psor:           [8:5,3];
         copied:         [9,f_or_t];
         level_no:       [10,16];
         dirty:          [12,1];
    END

    TEMPLATE system_type IS
      CASE [0:3,1] IS
         WHEN 0 => [8,5] IS (generic, resl, obj_tab, instr,
                             ctxt_ds, process_ds, psor_ds, port_ds,
                             carr_ds, sro_ds, comm_sg, des_ctl,
                             refn_ctl);
         WHEN 1 => [8,5] IS (generic, resl, domain, res3, ctxt,
                             process, psor, port, carrier, sro,
                             type_def, res12, res13);
      END CASE
    END

    TEMPLATE type_des IS
         descr_type:     [0,1] IS "Type Descriptor",/;
         valid:          [0:2,f_or_t];
         kind:           [0:3,1] IS (public,private),/;
         Type_defn_AD:   [4,32] IS rad,/;
         copied:         [9,f_or_t];
         level_no:       [10,16];
         Typed_obj_AD:   [12,32] IS rad;
    END
```

APPENDIX D
SAMPLE DEBUG-432 SESSIONS

INTRODUCTION

This appendix contains log files of two sample debugging sessions. The first demonstrates the use of many commands in a single process environment; the second uses the RESUME, SELECT and EXAMINE commands in a multi-process environment.

```
?-RUN DEB432
SERIES III  432 SYSTEMS LEVEL DEBUGGER, V1.00

? -- let's first look at some startup sequences:
?
? -- assume that the 432 memory is in an uninitialized state, and that
? -- we wish to load the file prime.eod, and run it
?
?init                        -- initialize 432 memory
TOP OF MEMORY IS: 524287
?load prime.eod              -- load the file
?debug                       -- enable logical addressing, 432 I/O,
?                            -- and breakpoints
?start                       -- send a wake up to the lowest-numbered
?                            -- GDP
?--FROM 432:
iMAX 432 V1.00

                             -- operating system has signed on
?--FROM 432:
Process 1 started, coordinates: 7^83

    SRO size = 2504 bytes, OT size = 100 descriptors

Global heap SRO size = 180272 bytes

Processor 2 dispatching

Processor 1 dispatching

                              PRIME FACTOR

This program determines the prime factors of a user specified integer
between 2 and 100,000. To exit the program merely enter 0 as the user
specified integer.
```

```
enter integer :
?                           -- the prime program has signed on and
?                           -- is running
?
?-----------------------------------------------------------------------
?
?-- let's try another standard startup sequence, this time assuming
?-- that the 432 memory has been initialized, and that the top of
?-- memory is at !7FFFF.  Confirm this by using the VERSION command:
?
?base 16; suffix 16
?version                -- display (among other things) the top-of-memory
SERIES III  432 SYSTEMS LEVEL DEBUGGER, V1.00
TOM: !7FFFF      MODE: DEBUGGING + I/O      LOG FILE: :F1:DEMO2.LOG (ON)
?
?-- notice that the top of memory is displayed in the current output
?-- base (16)
?
?-- the startup sequence:
?
?init sys !7ffff           -- reset the hardware
?                          -- this also halts the GDPs
?load prime.eod            -- this just loads the object file
?debug                     -- enable logical addressing, start polling
?                          -- for 432 I/O and breakpoints
?ipc 1,0                   -- same as "start 1"
?--FROM 432:
iMAX 432 V1.00

Process 1 started, coordinates: 7^83

    SRO size = 2504 bytes, OT size = 100 descriptors

Global heap SRO size = 180272 bytes

Processor 1 dispatching

                              Processor 2 dispatching

PRIME FACTOR

This program determines the prime factors of a user specified integer
between 2 and 100,000. To exit the program merely enter 0 as the user
specified integer.


enter integer :
-----------------------------------------------------------------------
?-- SAVE and RESTORE
?
?init
TOP OF MEMORY IS: 7FFFF
?load prime.eod
?save !0 to !3ffff to prime.sav
```

```
?
?-- another startup sequence, using the memory image file produced by
?-- the SAVE command:
?
?init sys !7ffff
?restore prime.sav
?debug                  -- enable logical addressing, I/O, breakpoints
?start
?--FROM 432:
iMAX 432 V1.00

Process 1 started, coordinates: 7^83

    SRO size = 2504 bytes, OT size = 100 descriptors

Global heap SRO size = 180272 bytes

Processor 2 dispatching

Processor 1 dispatching

                              PRIME FACTOR

This program determines the prime factors of a user specified integer
between 2 and 100,000. To exit the program merely enter 0 as the user
specified integer.


enter integer :      %124
?--FROM 432:
124 = 2 * 2 * 31


enter integer :      %2000
?--FROM 432:
2000 = 2 * 2 * 2 * 2 * 5 * 5 * 5


enter integer :
?-----------------------------------------------------------------------
?-- MODE
?
?-- the debugger is accepting both debugger commands and 432 input,
?-- preceded by "%"
?
?-- display the current mode
?
?mode
DEBUGGING + I/O
?
?-- now change the mode to 432 I/O only
?-- notice that in I/O ONLY mode, debugger commands are illegal,
?-- and 432 input is not preceded by "%"
```

```
?I/O ONLY
127
127 is a prime integer


enter integer :     63
63 = 3 * 3 * 7


enter integer :
?
?---------------------------------------------------------------
?-- EXAMINING MEMORY
?
?-- examining memory using default templates:
?
?-- using the link map, find the address of any domain, say 7^68
?-- now display the domain:
?7^68
DOMAIN_AS
  5^  3    5^  1    5^  2    5^  4    5^  5    7^ 67    0^  0    0^  0
  0^  0    0^  0    0^  0    7^ 5F    7^ 62    7^ 63    7^ 66    7^ 60
?-- notice that the debugger has found the type and used a default
?-- template to display the list of access descriptors in the domain
?
?-- again, find the coordinates of a process object in the link map,
?-- and display the process access segment:
?
?7^83
PROCESS_AS
STATUS:      9   [NOT_BOUND  NOT_FAULTED]
PROC_DS:       7^ 87
CURR_CTXT:     7^ 85
PGLOB_AS:      7^ 84
LOC_OBJ_TAB:   0^  0
PROC_CARR:     0^  0
DISPATCH_PT:   0^  0
SCHEDULE_PT:   0^  0
FAULT_PORT:    0^  0
CUR_MSG:       0^  0
CUR_PORT:      0^  0
CUR_CARR:      0^  0
SURR_CARR:     0^  0
?
?-- the default can be overridden by including a template name:
?
?7^83:ad                 -- display the 1st access descriptor in the object
  7^ 87
?
?-- a template may be repeated:
?
?7^83:ad len 3
  7^ 87
  7^ 85
  7^ 84
?
```

```
?-- an offset may be used.  For example, to display the 3rd AD:
?
?7^83.2:ad
  7^ 84
?
?-- in an access list, the offset "2" is interpreted as the 3rd AD
?-- in a data segment, the offset is interpreted as a byte offset
?7^83.1:ad
  7^ 85
?7^85
CONTEXT_AS
CTXT_DS:      7^ 86
CONST_DS:     0^  0
PREV:         0^  0
MSG:          0^  0
CURR_CTXT:    0^  0
EAS_1:        7^ 7D
EAS_2:        0^  0
EAS_3:        0^  0
DOMAIN:       7^ 7A
?7^83.0:ad                  -- this is the process data segment
  7^ 87
?7^87                       -- display the process DS
PROCESS_DS
         0       10       10        0        0        0        0        0
         0        0        0        0        1        0        0        0
     OFFFF      3E8        9        0        0        0        0        0
         0        0        0        0        0        0        0        0
?7^87.2:b16                 -- the offset is interpreted as 2 bytes
10
?
?-- to use a byte offset in an access object, use a "!" instead of "."
?
?7^83
PROCESS_AS
STATUS:       9   [NOT_BOUND  NOT_FAULTED]
PROC_DS:        7^ 87
CURR_CTXT:      7^ 85
PGLOB_AS:       7^ 84
LOC_OBJ_TAB:    0^  0
PROC_CARR:      0^  0
DISPATCH_PT:    0^  0
SCHEDULE_PT:    0^  0
FAULT_PORT:     0^  0
CUR_MSG:        0^  0
CUR_PORT:       0^  0
CUR_CARR:       0^  0
SURR_CARR:      0^  0
?7^83.2:ad
  7^ 84
?7^83!8:ad
  7^ 84
?
```

```
?-- examine the I/O buffer, which is at offset 16#14# in the processor
?-- access object:
?1^1.14:ad
  5^ 31
?-- display the I/O buffer using the template "mem", which displays
?-- data in both hex and ASCII:
?
?5^31:mem
  0  0  1  0  0  0 84  0 65 6E 74 65 72 20 69 6E   '........enter in'
 74 65 67 65 72 20 3A 20 20 20 20 20 0D 0A 74 65   'teger :     ..te'
 72 20 30 20 61 73 62 79 74 65 73 0D 0D 0A 30 30   'r 0 asbytes...00'
 20 64 65 73 63 72 69 70 74 6F 72 73 0D 0D 0A 20   ' descriptors... '
?
?-- to display more of the buffer, repeat the template:
?
?5^31:mem len 2
  0  0  1  0  0  0 84  0 65 6E 74 65 72 20 69 6E   '........enter in'
 74 65 67 65 72 20 3A 20 20 20 20 20 0D 0A 74 65   'teger :     ..te'
 72 20 30 20 61 73 62 79 74 65 73 0D 0D 0A 30 30   'r 0 asbytes...00'
 20 64 65 73 63 72 69 70 74 6F 72 73 0D 0D 0A 20   ' descriptors... '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
?
?-- fill the buffer with blanks:
?
?5^31.8:mem := 5^31.48
?5^31:mem
  0  0  1  0  0  0 84  0 20 20 20 20 20 20 20 20   '........        '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20   '                '
?
?-------------------------------------------------------------------
?-- NAMES --
?
?-- names may be defined at debug time.
?-- they are of 4 types: BREAK, INTEGER, REFERENCE, and TEMPLATE
?
?-- the names are stored in the directory
?-- the include file DEB432.TEM defined a basic set of templates.  These
?-- are in the directory:
?
?dir template
AD             TEMPLATE   END
AS             TEMPLATE   END
B16            TEMPLATE   END
B32            TEMPLATE   END
B8             TEMPLATE   END
BS             TEMPLATE   END
CH             TEMPLATE   []END
CONTEXT_AS     TEMPLATE   CTXT_DS; CONST_D#; PREV; MSG; CURR_CT#;
CONTEXT_DS     TEMPLATE   CTXT_ST#; SP; INST_ID#; IP; TRACE_I#; TRACE_I#;
```

```
DESCR         TEMPLATE   [[[]]]END
DS            TEMPLATE   END
DUMP          TEMPLATE   END
EXTRACT       TEMPLATE   END
FLT           TEMPLATE   END
FREE          TEMPLATE   DESCR_T#; FREE_IN#; END
F_AREA        TEMPLATE   FAULT_O#; PRE_IP; POST_IP; PRE_SP; POST_SP;
F_OR_T        TEMPLATE   END
HEADER        TEMPLATE   DESCR_T#; FREE_IN#; END_IND#; FAULT_L#;
INTERCONNECT  TEMPLATE   DESCR_T#; VALID; WINDOWE#; BASE_AD#; LENGTH_;
MEM           TEMPLATE   END
PFLT          TEMPLATE   END
PROCESS_AS    TEMPLATE   STATUS; PROC_DS; CURR_CT#; PGLOB_A#; LOC_OBJ#;
PROC_STAT     TEMPLATE   END
PSORFLT       TEMPLATE   END
RAD           TEMPLATE   END
RAS           TEMPLATE   END
REFINE        TEMPLATE   DESCR_T#; VALID; BASE_TY#; SYS_TYP#; BYPASS;
STORAGE       TEMPLATE   TYPE; VALID; BASE_TY#; SYS_TYP#; ALLOCAT#;
SYSTEM_TYPE   TEMPLATE   []END
TYPE_DES      TEMPLATE   DESCR_T#; VALID; KIND; TYPE_DE#; COPIED;
?
?-- a detailed description of each can be obtained. E.g.
?
?dir ad
TEMPLATE AD IS
  [2:4, OC] IS OU:3, "^";
  [0:4, OC] IS OU:3;
END
?
?-- a REFERENCE variable can be defined to save an address-template
?-- pair.  If the template name is omitted, the default will be used.
?
?-- for example, if we are going to look at the I/O buffer often,
?-- define a reference:
?
?buff is 5^31:mem
?
?-- display the reference:
?
?dir buff
BUFF IS 5^31:MEM
?
?-- a REFERENCE variable can be save the address of the process object,
?-- which is useful when setting breakpoints
?
?proc is 7^83
?
?-- let's also name the addresses of 3 instruction segments:
?
?main is 7^70
?get_num is 7^6d
?headr is 7^6c
?
```

```
?-- and name the domain "prompt" which contains the instruction
?-- segments "get_num" and "header" defined above
?
?prompt is 7^6f
?
?dir reference
GET_NUM       REFERENCE  [ IS 7^6D]
HEADR         REFERENCE  [ IS 7^6C]
MAIN          REFERENCE  [ IS 7^70]
PROC          REFERENCE  [ IS 7^83]
PROMPT        REFERENCE  [ IS 7^6F]
?
?-- integers are useful for naming the offsets of instructions within
?-- an instruction segment
?
?line_57:integer :=03a2
?
?dir integer
LINE_57       INTEGER    := 3A2
?
?-----------------------------------------------------------------------
?-- DEFINE AND HIT A BREAKPOINT
?
?init sys !7ffff
?debug prime.eod
?
?-- now set a breakpoint at the start of the main procedure,
?-- using symbols defined above:
?
?ba main of proc
?
?-- notice the default name and bit offset
?dir break
BO            BREAK      BA 7^70.50 OF 7^83
?
?start
?--FROM 432:
iMAX 432 V1.00

Process 1 started, coordinates: 7^83

    SRO size = 2504 bytes, OT size = 100 descriptors

Global heap SRO size = 180272 bytes

Processor 2 dispatching

Processor 1 dispatching

BO. BREAK AT: 7^70.50 OF 7^83
?
?-----------------------------------------------------------------------
?-- CC and CP
?
?-- now that the first break has been reached, CC and CP are defined.
?-- CP can be used to look at the current process:
```

```
?cp
PROCESS_AS
STATUS:       0C009   [NOT_BOUND  NOT_FAULTED]
PROC_DS:        7^ 87
CURR_CTXT:      8^ 0C
PGLOB_AS:       7^ 84
LOC_OBJ_TAB:    2^  8
PROC_CARR:      7^ 82
DISPATCH_PT:    6^ 15
SCHEDULE_PT:    0^  0
FAULT_PORT:     0^  0
CUR_MSG:        0^  0
CUR_PORT:       6^ 20
CUR_CARR:       7^ 82
SURR_CARR:      0^  0
?
?-- CC is the user's context (which reached a breakpoint)
?-- notice that the value of CC is not the same as CP.CURRENT_CONTEXT
?
?dir cc
CC IS 8^8
?
?cc
CONTEXT_AS
CTXT_DS:      8^  9
CONST_DS:     7^ 71
PREV:         8^  5
MSG:          8^  7
CURR_CTXT:    8^  8
EAS_1:        0^  0
EAS_2:        0^  0
EAS_3:        0^  0
DOMAIN:       7^ 72
?
?-----------------------------------------------------------------------
?-- USING THE DOT (.) OPERATOR WITH TEMPLATE FIELD NAMES
?
?-- invade the current process data structure:
?
?cp.
STATUS:       0C009   [NOT_BOUND  NOT_FAULTED]
PROC_DS:        7^ 87
CURR_CTXT:      8^ 0C
PGLOB_AS:       7^ 84
LOC_OBJ_TAB:    2^  8
PROC_CARR:      7^ 82
DISPATCH_PT:    6^ 15
SCHEDULE_PT:    0^  0
FAULT_PORT:     0^  0
CUR_MSG:        0^  0
CUR_PORT:       6^ 20
CUR_CARR:       7^ 82
SURR_CARR:      0^  0
```

```
?
?-- now the field names of "CP" are available:
?
?curr_ctxt.
CTXT_DS:      8^ 0D
CONST_DS:     5^ 18
PREV:         8^ 0A
MSG:          0^  0
CURR_CTXT:    8^ 0C
EAS_1:        7^ 83
EAS_2:        5^  5
EAS_3:        7^ 84
DOMAIN:       5^  5
?
?-- the "." after curr_ctxt has made the field names of CC available:
?
prev
  8^ 0A
?
?-- the path we have followed to get the value of "prev" is:
?
path
(7^83:PROCESS_AS).CURR_CTXT(8^0C)
?
?-- now just use a "." to invade the "prev" field. "prev" is the
?-- access descriptor of the calling context.
?
?.
CTXT_DS:      8^ 0B
CONST_DS:     7^ 71
PREV:         8^  8
MSG:          0^  0
CURR_CTXT:    8^ 0A
EAS_1:        7^ 72
EAS_2:        0^  0
EAS_3:        0^  0
DOMAIN:       7^ 72
?
?-- the path has changed:
?
?path
(7^83:PROCESS_AS).CURR_CTXT.PREV(8^0A)
?
?-- the context data segment of "prev" (8^0a) is:
?
?ctxt_ds
  8^ 0B
?
?-- now move back one level on the path:
?
?back
8^0C:CONTEXT_AS
?
```

```
?-- the context data segment of this context (8^0c) is:
?
?ctxt_ds
  8^ 0D
?
?path
(7^83:PROCESS_AS).CURR_CTXT(8^0C)
?
?-- use OUT to return to the original name scope:
?
?out
?path
ERR 177: NO PATH TO PRINT:  DST IS EMPTY
?
?-----------------------------------------------------------------------
?-- PROCEDURE ENTRY AND EXIT BREAKPOINTS
?
stack
CONTEXT            INSTRUCTION
8^8                7^70.50
8^5                7^74.127
8^1                7^7B.1A9
?
?-- we can trace all calls to and returns from a particular
?-- instruction segment or domain:
?
?be prompt                 -- break on entry to domain "prompt"
?bx headr                  -- break on exit from procedure "header"
?
?dir break
B0           -BREAK      BA 7^70.50 OF 7^83
B1            BREAK      BE 7^6F OF 7^83
B2            BREAK      BX 7^6C OF 7^83
?
?resume
?
B1. BREAK ENTER (DOM: 7^6F): 7^6C.50 OF 7^83
?
?-- we are at the entry to the first procedure in domain "prompt",
?-- which is "header"
?
?resume
?--FROM 432:
                              PRIME FACTOR

This program determines the prime factors of a user specified integer
between 2 and 100,000. To exit the program merely enter 0 as the user
specified integer.

B2. BREAK EXIT: 7^6C.0A3F OF 7^83
?
?-- now we are at the return from the procedure "header"
?
```

```
?-- set another breakpoint in the procedure "get_num",
?-- just after the user is prompted for a number:
?
?ba get_num.line_57
?resume
?
B1. BREAK ENTER (DOM: 7^6F): 7^6D.50 OF 7^83
?resume
?--FROM 432:


enter integer :     B3. BREAK AT: 7^6D.3A2 OF 7^83
?
?deactivate b1
?deactivate b2
?dir break
B0             BREAK      BA 7^70.50 OF 7^83
B1            *BREAK      BE 7^6F OF 7^83
B2            *BREAK      BX 7^6C OF 7^83
B3            -BREAK      BA 7^6D.3A2 OF 7^83
?----------------------------------------------------------------------
?-- BREAK ON CALL AND RETURN
?
?-- we can produce a trace of calls and returns by leaving
?-- the bo call and bo ret breakpoints set, and successively
?-- "resuming" the process
?
?bo call
?bo ret
?resume
?
B4. BO CALL: 7^66.50 OF 7^83
?resume
?
B4. BO CALL: 7^63.50 OF 7^83
?resume
?
B4. BO CALL: 7^53.50 OF 7^83
?resume
?
B4. BO CALL: 5^0E3.50 OF 7^83
?resume
?
B5. BO RET: 5^0E3.0F6 OF 7^83
?stack
CONTEXT                INSTRUCTION
8^1C                   5^0E3.0F6
8^18                   7^53.639
8^14                   7^63.38A
8^10                   7^66.268
8^0C                   7^6D.43F
8^8                    7^70.26A
8^5                    7^74.127
8^1                    7^7B.1A9
```

```
?------------------------------------------------------------------------
?-- BREAK AFTER EACH INSTRUCTION
?
?-- we can single-step through an instruction segment using
?-- the bo inst breakpoint.
?-- recall that we must first lift the bo ret and bo call
?-- breakpoints
?
?deactivate b4
?deactivate b5
?step: bo inst
?resume
?
STEP. BO INST: 7^53.639 OF 7^83
?resume
?
STEP. BO INST: 7^53.670 OF 7^83
?resume
?
STEP. BO INST: 7^53.69E OF 7^83
?resume
?
STEP. BO INST: 7^53.6CC OF 7^83
?dir break
BO             BREAK      BA 7^70.50 OF 7^83
B1            *BREAK      BE 7^6F OF 7^83
B2            *BREAK      BX 7^6C OF 7^83
B3             BREAK      BA 7^6D.3A2 OF 7^83
B4            *BREAK      BO CALL OF 7^83
B5            *BREAK      BO RET OF 7^83
STEP          -BREAK      BO INST OF 7^83
?------------------------------------------------------------------------
?deactivate step
?resume
?%1234
?--FROM 432:
1234 = 2 * 617


enter integer :     B4. BREAK AT: 7^6D.3A2 OF 7^83
?
?dir break
BO             BREAK      BA 7^70.50 OF 7^83
B1            *BREAK      BE 7^6F OF 7^83
B2            *BREAK      BX 7^6C OF 7^83
B3            -BREAK      BA 7^6D.3A2 OF 7^83
B4            *BREAK      BO CALL OF 7^83
B5            *BREAK      BO RET OF 7^83
STEP          *BREAK      BO INST OF 7^83
?------------------------------------------------------------------------
?-- ACTIVATE
?
?-- a breakpoint which was lifted can be reactivated:
?
?activate b4
```

```
?resume
?
B4. BO CALL: 7^66.50 OF 7^83
?resume
?
B4. BO CALL: 7^63.50 OF 7^83
?
?----------------------------------------------------------------------
?-- THE CALL STACK
?
?-- the call stack can be displayed
?
?stack
CONTEXT            INSTRUCTION
8^14               7^63.50
8^10               7^66.268
8^0C               7^6D.43F
8^8                7^70.26A
8^5                7^74.127
8^1                7^7B.1A9
?dir cc
CC IS 8^14
?
?-- CC now has the value of the bottom of the stack, 8^14, the context
?-- which reached the breakpoint.  To move up the call stack, to th
?-- context which called 8^14, use UP:
?
?up;dir cc
CC IS 8^10
?
?-- move to the first context in the process:
?
?top;dir cc
CC IS 8^1
?
?-- move down the call stack:
?
?down;dir cc
CC IS 8^5
?
?-- move back to the breakpointed context:
?
?botm;dir cc
CC IS 8^14
?----------------------------------------------------------------------
?exit
```

```
-RUN DEB432
SERIES III  432 SYSTEMS LEVEL DEBUGGER, V1.0

?-- the following session demonstrates the use of the EXAMINE,
?-- SELECT, and RESUME commands in a multiple process environment
?
?include :f1:deb432.tem
?init
TOP OF MEMORY IS: 7FFFF
?
?-- test.eod is a small 3-process program.  each process simply
?-- increments a variable and prints it
?
?debug test.eod
?
?-- use an include file to set a breakpoint at the first instruction
?-- in the "increment number and print" procedure of each process
?
?include :f1:breaks.inc list
?
?-- define some now some useful references:
?p1 is 9^16; p2 is 9^2b; p3 is 9^40;
?i1 is 9^47; i2 is 9^49; i3 is 9^4b
?-- and set the breaks:
?ba i1 of p1;ba i2 of p2;ba i3 of p3;
?
?start
?--FROM 432:
iMAX 432 X0.00-002
Process 1 started, coordinates: 9^16
    SRO size = 2504 bytes, OT size = 100 descriptors
Process 2 started, coordinates: 9^2B
    SRO size = 2504 bytes, OT size = 100 descriptors
Process 3 started, coordinates: 9^40
    SRO size = 2504 bytes, OT size = 100 descriptors
Global heap SRO size = 198032 bytes
Processor 2 dispatching
Processor 1 dispatching
B0. BREAK AT: 9^47.50 OF 9^16
?
B1. BREAK AT: 9^49.50 OF 9^2B
?
B2. BREAK AT: 9^4B.50 OF 9^40
?
?-- each process has reached a breakpoint.
?-- use the EXAMINE command to display the set of breakpointed
?-- processes:
?
examine
    PROCESS             CONTEXT
CP: 9^16                10^8
    9^2B                11^8
    9^40                12^8
```

```
?
?-- notice that the reference variables CP and CC have been defined
?-- the current process is the first process to reach a breakpoint
?
dir cp;dir cc
CP IS 9^16
CC IS 10^8
?
?-- now resume the current process, and look at the value
?-- of CP and CC while the process is executing:
?
?remove b0
?resume
?--FROM 432:
Process 1: 00000001
Process 1: 00000002
Process 1: 00000003
Process 1: 00000004
Process 1: 00000005
DEBUGGING ONLY
?
?-- (432 output was stopped with a CONTROL-C)
?
?dir cc;dir cp
CC IS 0^0
CP IS 9^16
?
?-- CC has is 0^0 whenever CP is executing
?-- CP retains its value until it is changed with a SELECT command
?
?-- now set a breakpoint in the current process
?
ba i1.0f0 of p1
?DEBUGGING + I/O
?--FROM 432:
Process 1: 00000006
B3. BREAK AT: 9^47.0F0 OF 9^16
?
?-- notice that CC is reassigned the value of the breakpointed context
?
?dir cp;dir cc
CP IS 9^16
CC IS 10^8
?
?-- now select a new process
?
?select p2
?
?-- note that cp has changed:
?
?examine
    PROCESS             CONTEXT
    9^16                10^8
CP: 9^2B                11^8
    9^40                12^8
```

```
?
?-- set breakpoints in the "increment number and print" loop of
?-- processes 9^2b and 9^40
?
ba i2.0f0 of p2;ba i3.0f0 of p3
?resume all
?--FROM 432:
Process 1: 00000007
B4. BREAK AT: 9^49.0F0 OF 9^2B
?
B3. BREAK AT: 9^47.0F0 OF 9^16
?
B5. BREAK AT: 9^4B.0F0 OF 9^40
?
?-- CP remains 9^2b (p2)
?
?examine
    PROCESS             CONTEXT
CP: 9^2B                11^8
    9^16                10^8
    9^40                12^8
?exit
```

INTRODUCTION

This appendix explains what to do when you think a fault has occurred when debugging:

- How to know when a fault has occurred
- How to find the faulting instruction in a listing
- How to display the appropriate fault information area
- How to use the Fault Decoding tables
- How to know when an unannounced fault has occurred
    - Unannounced processor level faults
    - Unannounced process level faults
- How to quickly recognize the most common faults

HOW TO KNOW WHEN A FAULT HAS OCCURRED

Usually when a fault occurs, DEBUG-432 catches it and prints a message similar to the breakpoint announcement message. For example, the message

```
    FAULT AT: 4^74.311 OF 4^44
```

announces that a fault was caused by the instruction at bit offset 311 in instruction segment 4^74 of process 4^44.

The debugger does not announce either processor level faults or faults (such as the Storage Claim Underflow Fault) that occur while it is processing another fault. Unannounced faults are described below.

HOW TO FIND THE FAULTING INSTRUCTION IN A LISTING

With the instruction segment coordinates (4^74) and the bit offset (311) of the faulting instruction in that segment, you can find the text for the faulting instruction in a an object code listing.

Run the linker with the "OBJECTMAP" directive, and scan the linker listing for the faulting instruction segment coordinates. You should find information about that segment, including its name. Backing up in the linker listing, find the name of the module that contains the faulting instruction segment. Look in the instruction listing generated by the compiler for this module. Find the instructions listed for the faulting instruction segment (whose name you found in the linker listing). The faulting instruction is identified by its bit offset.

Look at the instruction you've just found and the ones preceding it, and examine the original source code that was compiled to generate these instructions. More often than not, the cause of the fault is clear at this point. If not, you can track down more information about the fault as described in the next sections.

It is often useful to reinitialize the system, reload memory, set a breakpoint at or before the faulting instruction, and start again. When you reach the breakpoint, you can examine the data involved in the instruction to see if anything looks amiss.

HOW TO DISPLAY THE APPROPRIATE FAULT DATA AREA

Two fault data areas that may be useful are the processor fault data area in each processor data segment and the context fault data area in each process data segment. Process level fault information is found in the context fault data area.

The file DEB432.TEM includes the templates FLT, PFLT, and PSORFLT. FLT and PFLT can be applied directly to a process access segment to display the context and process fault data areas, respectively, in the associated process data segment. PSORFLT can be applied directly to a processor access segment to display the fault information area in the associated processor data segment. Each of these templates automatically locates the appropriate data segment from the given access segment and uses the appropriate displacement into that data segment to display the fault area.

For example, to display the context fault information area associated with the process 6^OB2, use the command "6^OB2:FLT". To display the processor fault information area for processor 1, use the command "1^1:PSORFLT".

USING FAULT TABLES

The tables at the end of this chapter will help you interpret the data in the fault information area you have displayed. The remainder of this section describes each field in relation to other information you may be able to gather about the fault.

The faulted instruction object DAI field identifies the instruction segment in which the fault occurred. The debugger's fault announcement message gives you the same information in a more intelligible form, but this field may be useful if the fault is unannounced, or if you want to verify the debugger's announcement message. Shift it right 2 bits (divide by 4) to get an index (in access descriptors) into the current domain. The access descriptor at this location in the domain should identify the instruction segment.

The Pre-Instruction Instruction Pointer and Post-Instruction Instruction Pointer fields identify the faulting instruction and the one following it. The post IP may not be correct if the faulting instruction has not been completely decoded. The debugger's fault announcement message also gives you the pre IP, but the pre and post IP fields in the fault information area may be useful if the fault is unannounced, or if you want to verify the debugger's announcement message.

The Pre-Instruction Stack Pointer, Post-Instruction Stack Pointer, and the Fault Status fields are rarely useful.

The Faulted Operator ID# field usually identifies the operator in the instruction which caused the fault unless the faulting instruction has not been completely decoded. Operator ID values and the operators they identify are summarized in the "Data Operator" table near the end of this appendix.

The Fault Code field tells what the fault is, and sometimes gives a bit more information. The Fault Decoding tables given later contain most of the information needed to decode the fault code. Take the following steps:

1. Look in the "Fault Types" section, below; use the TTTT in the fault code to determine the fault class.

2. For classes 1,2,3, and 9, no more information is available.

3. For class 0, look up the faulting operator in the Type 0 Fault List and use the LL EEEE bits in the fault code to identify the particular fault.

4. Fault class 4 enumerates two different faults: the Test System Type and Object Table Entry type faults. The KKKKK bits in the fault code give the DESIRED value of either the system type, or the object table entry type and information. The actual values (i.e., the values which caused the fault) are not available.

5. For classes 5 through 8 (memory access faults), the AAA, W, and SSSS bits give more information, as described in the section, "Types 5,6,7,8 Faults."

The Fault Access Selector and Fault Displacement fields are mainly useful for fault classes 5, 6, 7, 8, and may or may not be valid at other times. The section describing these types of faults tells you when these fields are meaningful.

HOW TO KNOW WHEN AN UNANNOUNCED FAULT HAS OCCURRED

The difficulty with unannounced faults is you don't know immediately that one has occurred. Suspect a fault if a processor or a process seems to be doing nothing for a long while. To confirm your suspicion, you need to know the coordinates of the processors and processes in your system. The processor coordinates are always 1^1 for processor 1, 1^2 for processor 2, 1^3 for processor 3, etc. (Processor 1 resides in slot 1, processor 2 in slot 2, etc.) With this knowledge, make the following checks to find out what, if anything, has faulted.

When a processor faults, a red light lights up on the GDP board of the faulted processor. If you can't see the GDP boards, you can also know whether a processor has faulted by displaying the fault area in its processor data segment. If the fault area is all zeroes, the processor has not faulted; otherwise, it probably has. For example, to determine whether processor 2 has faulted, display the fault area in the processor data segment with the command "1^2:PSORFLT".

You can know whether a process has faulted by displaying it. Its status will be either faulted or not faulted. If it is faulted, display the fault area in its process data segment. For example, if process 4^44 has faulted, display the context fault area in the process data segment with the command 4^44:FLT.

UNANNOUNCED PROCESSOR LEVEL FAULTS

When the debugger announces a fault, it tells you the location of the instruction which caused the fault. When a fault such as a processor level fault occurs unannounced, you have to find the faulting instruction yourself. For processor level faults, there may or may not have been a process executing at the time of the fault. If a process was executing, the coordinates of its process carrier should be in the "current process carrier" field in the processor access segment (offset = 1 access descriptor). For example, if processor 2 has faulted, display the process carrier coordinates with the command "1^2.1:AD". If these coordinates are 0^0, there was no process executing at the time of the fault so you can not find a faulting instruction segment. (You have to rely solely on the information you can find in the fault data area.)

Once you have the process carrier coordinates, obtain the following information:

1. Find the process access segment coordinates in the "carried object" field in the process carrier (offset=9 access descriptors).

2. Find the current context access segment coordinates in the "current context" field in the process access segment (offset=1 access descriptor).

3. Find the current domain coordinates in the domain field in the current context access segment (offset=8 access descriptors).

Display the processor fault data area if you have not already done so. The faulted object index field in the fault data area is used to identify the faulting instruction segment. Shift the faulted object index right 2 bits (divide by 4) to get an offset (in access descriptors) into the current domain found in step 3. Find the coordinates of the faulting instruction segment at this offset in the current domain. The Pre-Instruction Instruction Pointer field in the fault data area should be the bit offset into this instruction segment of the instruction which was executing at the time of the fault (see above).

UNANNOUNCED PROCESS LEVEL FAULTS

When a process level fault goes unannounced, it is almost always because either the process stack SRO is too small (Storage Claim Underflow Fault) or the process local object table is too small (Object Descriptor Exhaustion Fault). Either of these conditions causes the fault handling code in the 432 to fault when it attempts to handle the fault, and the debugger cannot be notified. Section 6 explains how to identify these faults. If some other process level fault goes unannounced, seek help from Intel.

HOW TO QUICKLY RECOGNIZE TWO COMMON FAULTS

This section describes two of the most common faults and their probable causes. Note that the fault codes given here are not unique since many bits in the fault code are unpredictable. For example, although 3AOE is given here for the Storage Claim Underflow Fault, the fault code 7AOE, for example, might also indicate the same fault. This list is only a beginning. You may want to add to it based on your own experience.

Fault Code: 309F (Descriptor Type Fault)

This fault code is almost always caused by an unresolved reference in your link (i.e., the faulting instruction attempted to access an object which was not linked in by the linker). To find the unlinked object, first find the faulting instruction in a listing. Attempt to display each of the operands to the instruction. When you attempt to display the unresolve reference, the debugger will print a message such as "ERR 278: SECOND COORD. OF 6^131 IS BAD: O.T. ENTRY NOT VALID". Look up the coordinates of the access descriptor (in this case, 6^131) in your linker listing, and you'll probably find the unresolved reference.

Fault Code: 3A00 (Access Descriptor Validity Fault)

This fault code is almost always caused by attempting to access an object with a null access descriptor. This is commonly caused by passing a null access descriptor as an access parameter to a procedure which expects a valid one, or by attempting to access an object with an access variable which has not been initialized.

FAULT DATA AREA

The generic Fault Data Area is a 48-byte record organized as follows:

	Byte Displacement
First Fault Data Item	n+46 n+38
Second Fault Data Item	n+36 n+28
	n+26
	n+24
Fault Displacement	n+22
Fault Access Selector	n+20
Fault Code	n+18
Faulted Operator ID#	n+16
Processor Status	n+14
Process Status	n+12
Fault Status	n+10
Pre-Inst. Stack Pointer	n+8
Post-Inst. Stack Pointer	n+6
Pre-Inst. Instruction Pointer	n+4
Post-Inst. Instruction Pointer	n+2
Faulted Inst. Obj. DAI	n

Figure E-1. Fault Data Area

The Fault Data Area for context-, process-, and processor-level faults has the same organization (shown above). Process objects contain Fault Data Areas for context- and process-level faults. Processor objects contain Fault Data Areas for processor-level faults. The fields in the Fault Data Area are interpreted as follows:

Faulted Inst. Obj. DAI (Bytes n thru n+1)
Records the DAI (domain access index) for the instruction object in which the faulted instruction is located.

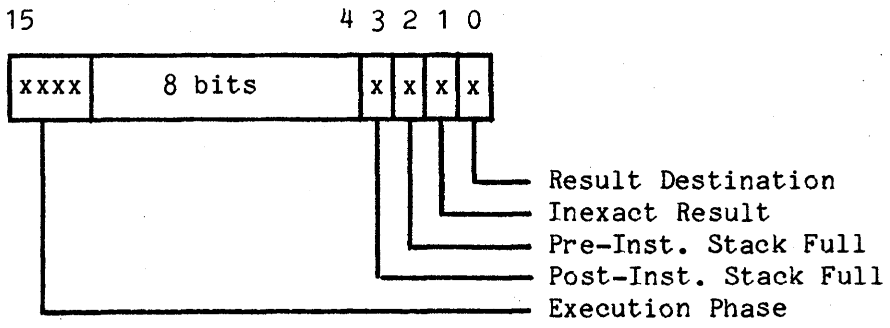
Post-Inst. Instruction Pointer (Bytes n+2 thru n+3)
Records the instruction pointer of the instruction physically following the instruction that caused the fault. If the fault occurred during instruction decoding, this field is undefined.

Pre-Inst. Instruction Pointer (Bytes n+4 thru n+5)
Records the instruction pointer of the instruction which caused the fault.

Post-Inst. Stack Pointer (Bytes n+6 thru n+7)
Records the operand stack pointer at the time the fault occurred. The actual stack pointer should be incremented by 2 if the Post-Inst. Stack Full bit in the Fault Status is 1.

Pre-Inst. Stack Pointer (Bytes n+8 thru n+9)
Records the operand stack pointer at the beginning of the instruction that caused the fault. The actual stack pointer should be incremented by 2 if the Pre-Inst. Stack Full bit in the Fault Status is 1.

Fault Status (Bytes n+10 thru n+11)
The Fault Status field has the following organization:



These fields are interpreted as follows:

Result Destination (Bit 0)
This bit records where the operand destination should have been:
0 - Destination was the operand stack
1 - Destination was in memory

Inexact Result (Bit 1)
This bit records whether the generated result was exact or inexact:

0 - exact
1 - inexact

Pre-Inst. Stack Full (Bit 2)
This bit records whether the 16-bit on-chip top of stack register was occupied at the beginning of the faulted instruction:

0 - empty
1 - occupied

Post-Inst. Stack Full (Bit 3)
This bit records whether the on-chip top-of-stack register was occupied when the instruction faulted:

0 - empty
1 - occupied

Execution Phase (Bits 12 - 15)
This 4-bit field records a value that indicates the phase of execution when the fault occurred. It is used to identify fault handling strategies in the more complex operators. A value of zero indicates that the instruction can be re-executed with no fault handling repair of data necessary.

Process Status (Bytes n+12 thru n+13)
This 16-bit field records the current process status at the time the fault occurred.

Processor Status (Bytes n+14 thru n+15)
This 16-bit field records the current processor status at the time the fault occurred.

Faulted Operator ID# (Bytes n+16 thru n+17)
If the fault occurred during instruction decoding, this field is zero. Otherwise, this field records the operator ID# of the faulted instruction.

Fault Code (Bytes n+18 thru n+19)
The Fault Code field contains a hardware-written 16-bit encoding that indicates the specific fault that occurred. The detailed encodings of this field are defined in subsequent sections of this chapter.

Fault Access Selector (Bytes n+20 thru n+21)
The interpretation of this field varies depending on the specific fault. See the following sections of this chapter for more details.

Fault Displacement (Bytes n+22 thru n+23)
The interpretation of this field varies depending on the specific fault. See the following sections of this chapter for more details.

Second Fault Data Item (Bytes n+28 thru n+37)

The value in this field depends on whether the fault is pre-operation or post-operation:

- If the fault is pre-operation, this field contains the value of source operand 2. Unused high-order bits are undefined.
- If the fault is post-operation, this field in not defined.

First Fault Data Item (Bytes n+38 thru n+47)

The value in this field depends on whether the fault is pre-operation or post-operation:

- If the fault is pre-operation, this field contains the value of source operand 1. Unused high-order bits are undefined.
- If the fault is post-operation, this field contains the value of the exceptional result. Unused high-order bits are undefined.

FAULT TYPES

Faults are categorized into nine general types as determined by bits 5 through 8 of the Fault Code field:

```
 15              8   7              0
+-----------------+-----------------+
| x x x x x x x T | T T T x x x x x |
+-----------------+-----------------+
```

In subsequent encoding diagrams in this chapter, the x values designate bits that are undefined for the particular fault type being described. The TTTT bits are used to encode the type of the fault that occurred. The remaining bits (designated above by x's) are used to further encode the specific fault.

The following list defines the TTTT encodings and gives a two letter mnemonic for the fault type. These mnemonics are used throughout this chapter.

TYPE	TTTT	MNEM	Faults
0	0000	FF	All other faults not named here
1	0001	SC	Index overflow (during scaling)
2	0010	DP	Displacement overflow during address development
3	0011	IP	Inst. pointer overflow during relative branch
4	0100	TS	Test system type or descriptor type faults
5	0101	SO	Segment overflow fault
6	0110	MO	Memory overflow fault (physical addr ≥ 2**24)
7	0111	RR	Read Rights fault
8	1000	WR	Write Rights fault
9	1001	TW	Destination Access Segment access rights fault

All faults of types 1 through 9 are process-level faults. Subsequent sections of this chapter describe the more detailed fault encodings for the nine fault types.

Type 0 Faults

Type 0 faults have the following bits defined in the Fault Code field:

```
 15             8   7            0
+-----------------+-----------------+
| x x x x x L L 0 | 0 0 0 x E E E E |
+-----------------+-----------------+
```

The LL bits encode the fault level as follows:

LL	Description
00	Context-Level Faults
01	Process-Level Faults (group 1)
10	Process-Level Faults (group 2)
11	Processor-Level Faults

The EEEE bits encode the specific fault within the level group.

The following Type 0 Fault List presents the type 0 faults in the order of their encoding. The encoding column of this table (and of other tables in the following sections) contains the LL EEEE bits if the type is 0 (FF).

TYPE 0 FAULT LIST

FAULTS	TYPE	ENCODING
		LL EEEE
Domain Error	0 (FF)	00 0000
Overflow	0 (FF)	00 0001
Underflow	0 (FF)	00 0010
Inexact	0 (FF)	00 0011
Invalid Class Fault	0 (FF)	00 0100
		LL EEEE
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001
Instruction Object Index Overflow Fault	0 (FF)	01 0010
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
Destination Delete Rights Fault	0 (FF)	01 0011
Race Condition Fault	0 (FF)	01 0011
Level Fault	0 (FF)	01 0100
Level Overflow Fault	0 (FF)	01 0100
Access Path Object Descriptor Type Faults	0 (FF)	01 0101
Entry Index Range Fault	0 (FF)	01 0101
Instruction Object Type Rights Fault	0 (FF)	01 0101
Odd Interconnect Descriptor Base Address Fault	0 (FF)	01 0101
Source Object Validity Fault	0 (FF)	01 0101
Surrogate Carrier Validity/Type Rights Fault	0 (FF)	01 0101
Access Segment Read Rights Fault	0 (FF)	01 0110
Context Parameters Size Faults	0 (FF)	01 0110
TCO Type Rights Fault	0 (FF)	01 0110

Odd Displacement Fault	0 (FF)	01 0110
Port Type Rights Fault	0 (FF)	01 0110
Processor Type Rights Fault	0 (FF)	01 0110
RCO Type Rights Fault	0 (FF)	01 0110
Return Level Fault	0 (FF)	01 0110
Source Object Access Rights Fault	0 (FF)	01 0110
TDO Validity Fault	0 (FF)	01 0110
Object Table Type Rights Fault	0 (FF)	01 0111
SRO Type Rights Fault	0 (FF)	01 0111
TDO Type Rights Fault	0 (FF)	01 0111
Clear Memory Size Fault	0 (FF)	01 1000
Type Fault	0 (FF)	01 1000
Carrier Lock Fault	0 (FF)	01 1001
Object Lock ID/Type Fault	0 (FF)	01 1001
Offset and Length Compatibility Fault	0 (FF)	01 1001
SRO Lock Fault	0 (FF)	01 1001
Port Lock Fault	0 (FF)	01 1010
Refinement Overflow Fault	0 (FF)	01 1010
Object Descriptor Exhaustion Fault	0 (FF)	01 1011
Storage Block Index Overflow Fault	0 (FF)	01 1101
Storage Claim Underflow Fault	0 (FF)	01 1110
Storage Block Fragmentation Fault	0 (FF)	01 1111
		LL EEEE-
Instruction Fetch Fault	0 (FF)	10 0000
Instruction Object Displacement Fault	0 (FF)	10 0000
		LL EEEE-
Bus Error	0 (FF)	11 0000
Process Level Objects Lock Fault	0 (FF)	11 0001
Process Lock Fault	0 (FF)	11 0001
PCO Lock Fault	0 (FF)	11 0011
Wakeup IPC Fault	0 (FF)	11 0100
Carrier Queued Fault	0 (FF)	11 0101

Types 1,2,3,9 Faults

These types have only the TTTT bits defined in the Fault Code field to distinguish them. Each fault type is thus a single fault.

Type 4 Faults

Type 4 faults have the following bits defined in the Fault Code field:

```
 15              8   7             0
+-----------------+-----------------+
| Z Q x x x x x 0 | 1 0 0 K K K K K |
+-----------------+-----------------+
```

The Z bit indicates whether the fault resulted with testing the system type or the object table entry type. The Z bit is defined as follows:

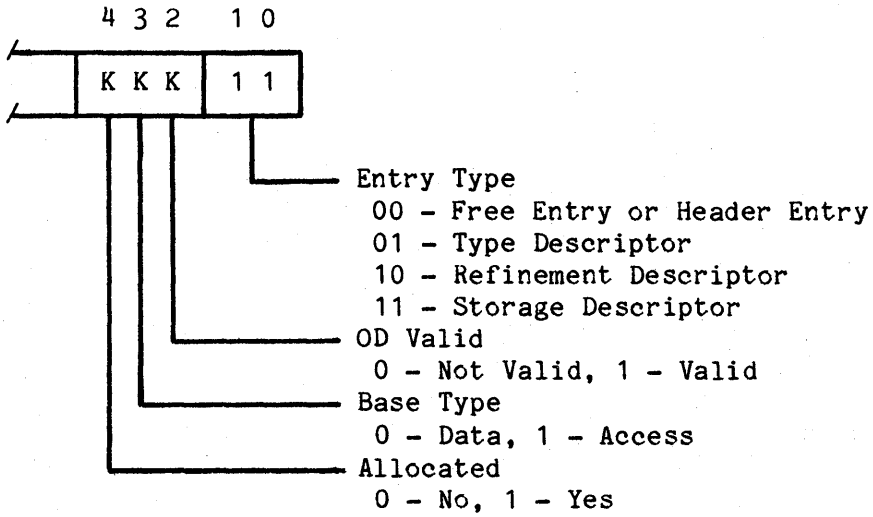
0 - OTE type test
1 - System type test

The Q bit indicates whether the fault is associated with object table qualification. It thus determines the meaning of the Fault Access Selector and Fault Displacement fields in the fault data area as follows:

0 - The fault did not occur during object table qualification and the Fault Access Selector and Fault Displacement fields contain the indices in the associated access descriptor.
1 - The fault occurred during object table qualification and the Fault Displacement field contains the directory index.

The Z bit determines two alternate interpretations of the KKKKK bits as follows:

Z=0 (fault because of object table entry type test). The KKKKK bits encode the expected values of the least-significant 5 bits of the object table entry (the actual values are unavailable). Their meanings are thus determined by the expected Entry Type of the object table entry. The following case is for a storage descriptor:



Z=1 (fault because of system type test). The KKKKK bits encode the expected value of the System Type field in the faulted object table entry (the actual values are unavailable):

KKKKK	SYSTEM TYPE
00000	Generic Access or Data Segment
00010	Domain Access Segment or Object Table Data Segment
00011	Instruction Data Segment
00100	Context Access or Data Segment
00101	Process Access or Data Segment
00110	Processor Access or Data Segment
00111	Port Access or Data Segment
01000	Carrier Access or Data Segment
01001	SRO Access or Data Segment
01010	TDO Access Segment or PCO Data Segment
01011	Type Control Data Segment
01100	Refinement Control Data Segment

The encoding column of the tables in the following sections contains the Z KKKKK bits if the type is 4 (TS).

Types 5,6,7,8 Faults

These faults have the following bits defined in the Fault Code field:

```
15             8  7              0
 x W A A A x x T | T T T x S S S S
```

These fault types are memory access faults. The W bit indicates whether the fault occurred on a read or write:

0 - Faulted on Read
1 - Faulted on Write

The AAA bits indicate the type of memory access that faulted:

AAA	TYPE OF ACCESS
0xx	Storage Address Space The storage segment being accessed is indicated by the SSSS bits. Displacement is given by the Fault Displacement field in the Fault Data Area.
100	Interconnect Address Space Displacement is given by the Fault Displacement field in the Fault Data Area.
101	Access Segment The access selector of the segment is given by the Fault Access Selector in the Fault Data Area.
111	Operand Stack Displacement is given by the Post-Inst. Stack Pointer field in the Fault Data Area.

The SSSS bits only have meaning if the AAA bits are encoded 0xx (i.e., the most-significant A bit is set). When this is the case, the SSSS bits encode the type of segment being accessed when the fault occurred:

SSSS	SEGMENT BEING ACCESSED
0000	Context Access Segment
0100	Object Table Directory
0101	Processor Access Segment
0111	Process Access Segment
1000	Instruction Object
1001	Context Data Segment
1010	Defining Domain
1011	Process Data Segment
1110	Data Segment Cache (The Fault Access Selector field contains the Access Selector of the segment).
1111	Object Table Cache (The Fault Access Selector field contains the directory index from the AD).

SSSS values 0001, 0010, 0011, 0110, 1100, and 1101 are undefined.

GENERAL FAULT GROUPS

The following faults can occur anywhere during the execution of an operator or sub-operation (which includes instruction decoding, process dispatching, binding etc.). These faults are not explicitly referenced in the later sections. The => symbol indicates that the group name preceding it stands for any of the possible faults that are listed after it. A group name is used in this table (and others in this chapter) by enclosing the name in angle brackets <like so>. This indicates that any of the possible faults of that named group are included.

FAULT GROUPS	TYPE	ENCODING
Memory Reference Faults =>		
Segment Bound Fault	5 (SO)	
Memory Overflow Fault	6 (MO)	
Read Rights Fault	7 (RR)	
Write Rights Fault	8 (WR)	
Bus Error	0 (FF)	11 0000
Invalid Class Fault	0 (FF)	00 0100
Instruction Fetch Fault	0 (FF)	10 0000
Data Segment Cache Qualification Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Type Fault	4 (TS)	0 10111
	4 (TS)	0 00110
	4 (TS)	0 11111
Object Table Cache Qualification Faults =>		
Object Descriptor Type Fault	4 (TS)	0 10111
Object System Type Fault	4 (TS)	1 00010
(Access) Segment Altered Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001
(Data) Segment Altered Faults =>		
<Data Segment Cache Qualification Faults>		

DATA OPERATOR FAULT GROUPS

FAULT GROUP	TYPE	ENCODING LL EEEE
Domain Error	0 (FF)	00 0000
Overflow	0 (FF)	00 0001
Underflow	0 (FF)	00 0010
Inexact	0 (FF)	00 0011

DATA OPERATORS

Table E-1 lists the data operator ids and associated mnemonics for the GDP. In the table, operators marked with ** do not have a unique operator ID; the compiler encodes them as either absolute branches or relative branches. Absolute branches have an operator ID of 254, while relative branches have an operator ID of 255. A conditional branch that is not taken has no operator ID.

Table E-1. Operator IDs and Mnemonics

1	movoc	51	inc_si	101	ptv_i
2	zro_c	52	dec_si	102	ntv_i
3	one_c	53	neg_si	103	cvt_i_si
4	sav_c	54	mul_si	104	cvt_i_o
5	and_c	55	div_si	105	cvt_i_tr
6	ior_c	56	rem_si	106	add_sr
7	xor_c	57	lss_si	107	add_sr_tr
8	eqv_c	58	leq_si	108	add_tr_sr
9	not_c	59	ptv_si	109	sub_sr
10	add_c	60	ntv_si	110	sub_sr_tr
11	sub_c	61	cvt_si_i	111	sub_tr_sr
12	inc_c	62	cvt_si_tr	112	mul_sr
13	dec_c	63	mov_o	113	mul_sr_tr
14	eql_c	64	zro_o	114	mul_tr_sr
15	neq_c	65	one_o	115	div_sr
16	eqz_c	66	sav_o	116	div_sr_tr
17	nez_c	67	and_o	117	div_tr_sr
18	lss_c	68	ior_o	118	neg_sr
19	leq_c	69	xor_o	119	abs_sr
20	cvt_c_so	70	eqv_o	120	eql_sr
21	mov_so	71	not_o	121	eqz_sr
22	zro_so	72	ext_o	122	leq_sr
23	one_so	73	ins_o	123	lss_sr
24	sav_so	74	sig_o	124	ptv_sr
25	and_so	75	add_o	125	ntv_sr
26	ior_so	76	sub_o	126	cvt_sr_tr
27	xor_so	77	inc_o	127	mov_r
28	eqv_so	78	dec_o	128	zro_r
29	not_so	79	mul_o	129	sav_r
30	ext_so	80	div_o	130	add_r
31	ins_so	81	rem_o	131	add_r_tr
32	sig_so	82	eql_o	132	add_tr_r
33	add_so	83	neq_o	133	sub_r
34	sub_so	84	eqz_o	134	sub_r_tr
35	inc_so	85	nez_o	135	sub_tr_r
36	dec_so	86	leq_o	136	mul_r
37	mul_so	87	lss_o	137	mul_r_tr
38	div_so	88	cvt_o_so	138	mul_tr_r
39	rem_so	89	cvt_o_i	139	div_r
40	eql_so	90	cvt_o_tr	140	div_r_tr
41	neq_so	91	add_i	141	div_tr_r
42	eqz_so	92	sub_i	142	neg_r
43	nez_so	93	inc_i	143	abs_r
44	lss_so	94	dec_i	144	eql_r
45	leq_so	95	neg_i	145	eqz_r
46	cvt_so_c	96	mul_i	146	leq_r
47	cvt_so_o	97	div_i	147	lss_r
48	cvt_so_tr	98	rem_i	148	ptv_r
49	add_si	99	leq_i	149	ntv_r
50	sub_si	100	lss_i	150	cvt_r_tr

151 mov_tr
152 zro_tr
153 sav_tr
154 add_tr
155 sub_tr
156 mul_tr
157 div_tr
158 rem_tr
159 neg_tr
160 sqt_tr
161 abs_tr
162 eql_tr
163 eqz_tr
164 leq_tr
165 lss_tr
166 ptv_tr
167 ntv_tr
168 cvt_tr_o
169 cvt_tr_i
170 cvt_tr_sr
171 cvt_tr_r
172 copy_ad
173 null_ad
174 amplify_rights
175 restrict_rights
176 create_pri_type
177 create_pub_type
178 retrieve_pub_type_rep
179 retrieve_type_rep
180 retrieve_type_def
181 create_rfn
182 create_typed_rfn
183 retrieve_rfn_obj
184 create_dseg
185 create_aseg
186 create_typed_seg
187 create_ad
188 inspect_ad
189 inspect_obj
190 lock_obj
191 unlock_obj
192 indiv_add_so
193 indiv_add_o
194 indiv_ins_so
195 indiv_ins_o
** br
** br_t
** br_f
196 br_ndirect
197 br_iseg
198 br_iseg_wo_trace
199 br_iseg_link
200 enter_aseg
201 enter_global_aseg
202 set_mode
203 call
204 call_msg
205 ret
206 send
207 receive
208 cond_send
209 cond_receive
210 sur_send
211 sur_receive
212 wait
213 read_prcs_clock
214 send_psor
215 bcst_psors
216 read_psor_status
217 mov_to_ict
218 mov_fm_ict

SUB-OPERATOR FAULT GROUPS

FAULT GROUPS	TYPE	ENCODING
Store Access Descriptor Faults =>		
Level Fault	0 (FF)	01 0100
Destination Delete Rights Fault	0 (FF)	01 0011
Object Qualification Faults =>		
Access Descriptor Validity Fault	0 (FF)	01 0000
Object Descriptor Fault	0 (FF)	01 0001
Object Descriptor Type Fault	4 (TS)	0 10111
	4 (TS)	0 11111
Descriptor Allocation Faults =>		
SRO Type Rights Fault	0 (FF)	01 0111
<Object Qualification Faults (SRO ASEG)>	4 (TS)	1 01001
<Object Qualification Faults (PSO DSEG)>	4 (TS)	1 01001
<Object Qualification Faults (Object Table DSEG)>	4 (TS)	1 00010
Object Descriptor Exhaustion Fault	0 (FF)	01 1011
Segment Allocation Faults =>		
SRO Lock Fault	0 (FF)	01 1001
Storage Block Index Overflow Fault (missing last block bit)	0 (FF)	01 1101
Storage Block Fragmentation Fault	0 (FF)	01 1110
Storage Claim Underflow Fault	0 (FF)	01 1111
Clear Memory Size Fault (Fault Access Selector contains the destination access selector)	0 (FF)	01 1000
Port Operation Faults =>		
<Object Qualification Faults (Carrier ASEG)>	4 (TS)	1 01000
<Object Qualification Faults (Carrier DSEG)>	4 (TS)	1 01000
<Object Qualification Faults (Port ASEG)>	4 (TS)	1 00111
<Object Qualification Faults (Port DSEG)>	4 (TS)	1 00111
Carrier Lock Fault	0 (FF)	01 1001
Wakeup IPC Fault	0 (FF)	11 0100
Port Lock Fault	0 (FF)	01 1010
Carrier Queued Fault	0 (FF)	11 0101
Context Qualification Faults =>		
<Object Qualification Faults (Context ASEG)>	4 (TS)	1 00100
<Object Qualification Faults (Context DSEG)>	4 (TS)	1 00100
<Object Qualification Faults (Domain)>	4 (TS)	1 00010
<Object Qualification Faults (Instruction)>	4 (TS)	1 00011
Process Binding and Qualification Faults =>		
<Object Qualification Faults (Process ASEG)>	4 (TS)	1 00101
<Object Qualification Faults (Process DSEG)>	4 (TS)	1 00101
Process Level Object Lock Fault	0 (FF)	11 0001
<Context Qualification Faults>		

NON-INSTRUCTION INTERFACE FAULTS

OPERATOR	TYPE	ENCODING
Initialization =>		
<Object Qualification Faults (Processor ASEG)>	4 (TS)	1 00110
<Object Qualification Faults (Obj. Table Directory)>	4 (TS)	1 00010
<Object Qualification Faults (Processor DSEG)>	4 (TS)	1 00110
<IPC Faults>		
IPC Faults =>		
<Object Qualification Faults (PCO)>	4 (TS)	1 01010
PCO Lock Fault	0 (FF)	11 0011
<IPC Faults>		
Idle =>		
<Delay Port Service Faults>		
Process Binding =>		
<Object Qualification Faults (Carrier ASEG)>	4 (TS)	1 01000
<Object Qualification Faults (Carrier DSEG)>	4 (TS)	1 01000
Process Lock Fault	0 (FF)	11 0001
<Process Qualification Faults>		
<Port Operation Faults>		
Process Selection =>		
<Delay Port Service Faults>		
<Object Qualification Faults (Carrier ASEG)>	4 (TS)	1 01000
<Object Qualification Faults (Carrier DSEG)>	4 (TS)	1 01000
<Port Operation Faults>		

OBJECT OPERATOR FAULTS

OPERATOR	TYPE	ENCODING
Copy Access Descriptor		
<Store Access Descriptor Faults>		
Null Access Descriptor		
Destination Delete Rights Fault	0 (FF)	01 0011
Amplify Rights		
TCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (TCO)>	4 (TS)	1 01011
Destination Access Segment Rights Fault	9 (TW)	
Source Object Validity Fault	0 (FF)	01 0101
Type Fault	0 (FF)	01 1000
Race Condition Fault (the access descriptor was changed before the amplified value is stored back)	0 (FF)	01 0011
Restrict Rights		
no explicit fault cases		
Create Public Type		
Create Private Type		
Destination Access Segment Rights Fault	9 (TW)	
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
TDO Validity Fault	0 (FF)	01 0110
TDO Type Rights Fault	0 (FF)	01 0111
<Object Qualification Faults (TDO)>	4 (TS)	1 01010
<Descriptor Allocation Faults>		
Level Faults	0 (FF)	01 0100
<Store Access Descriptor Faults>		
Retrieve Public Type Representation		
Source Object Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	4 (TS)	
<Store Access Descriptor Faults>		
Retrieve Type Representation		
TDO Validity Fault	0 (FF)	01 0110
Source Object Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	4 (TS)	
TDO Type Rights Fault	0 (FF)	01 0111
Type Fault	0 (FF)	01 1000
<Store Access Descriptor Faults>		
Retrieve Type Definition		
Source Object Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	4 (TS)	
<Store Access Descriptor Faults>		

OPERATOR	TYPE	ENCODING
Create Refinement		
Destination Access Segment Rights Fault	9 (TW)	
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
Source Object Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	4 (TS)	
Offset and Length Compatibility Fault	0 (FF)	01 1001
Refinement Overflow Fault	0 (FF)	01 1010
<Descriptor Allocation Faults>		
Level Fault	0 (FF)	01 0100
<Store Access Descriptor Faults>		
Create Typed Refinement		
RCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (RCO)>	4 (TS)	1 01100
Destination Access Segment Rights Fault	9 (TW)	
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
Source Object Validity Fault	0 (FF)	01 0101
Object Descriptor Type Fault	4 (TS)	
Type Fault	0 (FF)	01 1000
Offset and Length Compatibility Fault	0 (FF)	01 1001
Refinement Overflow Fault	0 (FF)	01 1010
<Descriptor Allocation Faults>		
Level Fault	0 (FF)	01 0100
<Store Access Descriptor Faults>		
Retrieve Refined Object		
RCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (RCO)>	4 (TS)	1 01100
Source Object Validity Fault	0 (FF)	01 0101
Type Fault	0 (FF)	01 1000
<Store Access Descriptor Faults>		
Create Data Segment		
Create Access Segment		
Destination Access Segment Rights Fault	9 (TW)	
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
<Descriptor Allocation Faults>		
<Segment Allocation Faults>		
<Store Access Descriptor Faults>		
Create Typed Segment		
TCO Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (TCO)>	4 (TS)	1 01100
Destination Access Segment Rights Fault	9 (TW)	
Pre-creation Destination Delete Rights Fault	0 (FF)	01 0010
<Descriptor Allocation Faults>		
<Segment Allocation Faults>		
<Store Access Descriptor Faults>		

OPERATOR	TYPE	ENCODING
Create Access Descriptor		
Object Table Type Rights Fault	0 (FF)	01 0111
<Object Qualification Faults (Object Table)>	4 (TS)	1 00010
Access Path Object Descriptor Type Faults	0 (FF)	01 0101
Inspect Access Descriptor		
no explicit fault cases		
Inspect Object		
Access Path Object Descriptor Type Faults	0 (FF)	01 0101
Lock Object		
<Object Qualification Faults (data segment)>		
Source Object Access Rights Fault	0 (FF)	01 0110
Unlock Object		
<Object Qualification Faults (data segment)>		
Source Object Access Rights Fault	0 (FF)	01 0110
Object Lock ID/Type Fault	0 (FF)	01 1001
Indivisibly Add Short Ordinal		
Indivisibly Add Ordinal		
Indivisibly Insert Short Ordinal		
Indivisibly Insert Ordinal		
no explicit fault cases		
Branch		
Branch True		
Branch False		
Instruction Pointer Overflow Fault	3 (IP)	
Instruction Object Displacement Fault	0 (FF)	10 0000
Branch Indirect		
Instruction Object Displacement Fault	0 (FF)	10 0000
Branch Intersegment		
Branch Intersegment without Trace		
Branch Intersegment and Link		
<Object Qualification Faults (Domain)>	4 (TS)	1 00010
<Object Qualification Faults (Instruction)>	4 (TS)	1 00011
Instruction Object Displacement Fault	0 (FF)	10 0000
Enter Access Segment		
Enter Global Access Segment		
Entry Index Range Fault	0 (FF)	01 0101
Access Segment Read Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (access segment)>		
Set Mode		
no explicit fault cases		

OPERATOR	TYPE	ENCODING
Call		
Call with Message		
<Object Qualification Faults (Domain)>	4 (TS)	1 00010
Instruction Object Type Rights Fault	0 (FF)	01 0101
<Object Qualification Faults (Instruction)>	4 (TS)	1 00011
Context Parameters Size Fault	0 (FF)	01 0110
Level Overflow Fault	0 (FF)	01 0100
Instruction Object Index Overflow Fault	0 (FF)	01 0010
<Descriptor Allocation Faults>		
<Storage Allocation Faults>		
<Context Qualification Faults>		
Instruction Object Displacement Fault	0 (FF)	10 0000
Return		
Return Level Fault	0 (FF)	01 0110
SRO Lock Fault	0 (FF)	01 1001
Level Overflow Fault	0 (FF)	01 0100
<Context Qualification Faults>		
Instruction Object Displacement Fault	0 (FF)	10 0000
Send		
Receive		
Conditional Send		
Conditional Receive		
Wait		
Port Type Rights Fault	0 (FF)	01 0110
<Port Operation Faults>		
Surrogate Send		
Surrogate Receive		
Surrogate Carrier Validity/Type Rights Fault	0 (FF)	01 0101
Port Type Rights Fault	0 (FF)	01 0110
<Port Operation Faults>		
Read Process Clock		
no explicit fault cases		
Send to Processor		
Broadcast to Processors		
Processor Type Rights Fault	0 (FF)	01 0110
<Object Qualification Faults (Processor ASEG)>	4 (TS)	1 00110
<Object Qualification Faults (PCO)>	4 (TS)	1 01010
Read Processor Status		
no explicit fault cases		
Move to Interconnect		
Move from Interconnect		
Odd Displacement Fault	0 (FF)	01 0110
Odd Interconnect Descriptor Base Address Fault	0 (FF)	01 0101
<Object Qualification Faults (Interconnect)>	4 (TS)	0 01100

APPENDIX F
DEBUG-432 ERROR MESSAGES AND POSSIBLE CAUSES

When DEBUG-432 detects an error, it sends a message to the user identifying the nature of the error. Errors are divided into two categories: user errors and internal debugger errors.

Internal errors indicate that a consistency check in the debugger has failed; the user should contact Intel. Internal errors are reported simply as numbers in the range 5000 through 5999.

User errors are prefixed with the string "ERR nnn: ", where nnn is the error number. Some user error messages contain text previously typed in by the user. This text is always surrounded by double quotes when the error is printed and is shown as an expression in angle brackets in the following list (e.g., <name>, <number>).

The rest of this appendix explains the possible user errors.

101 'MAY ONLY ASSIGN TO MEMORY REFERENCE'

The ":=" operator in the debugger means "modify memory". If the identifier to the left of the ":=" was not defined as a reference (i.e, using "is") it is illegal to use the value. For example:

```
?foo: integer := 5
?foo := foo + 1
```

causes the error, but the following commands do not:

```
?foo: integer := 5
?baz is 3^4:b16
?baz := foo
```

102 '"<name>" (TO LEFT OF ":=") IS NOT DEFINED'

If the user types in a command to modify memory, using a name to the left of the ":=", err 102 is displayed if that name is undefined.

103 'MEMORY WILL NOT BE MODIFIED'

The debugger views modifying 432 memory as a very important operation, and notifies the user when an apparent attempt to modify memory will not take place. This message is never displayed by itself, but only after another error message. Other error messages that can cause this one to be displayed are: 101, 102, 107, 153, 195, and 196.

104 'MUST HAVE EXPRESSION ON RHS OF ":="'

This is really a syntax error. It can happen only when the ":=" is the last symbol of the command. For example:

```
?foo is 3^4:b16
?foo :=
```

causes this error. Memory will not be changed.

105 'DESTINATION OF ":=" MUST BE A MEMORY REFERENCE'

The ":=" operator of the debugger means "modify memory". Although different expressions are syntactically legal to the left of the ":=", only those expressions which evaluate to an address template pair (i.e., a memory reference) may be used.

107 'RHS OF ":=" MUST BE A NUMBER OR MEMORY REFERENCE'

When modifying memory, the expression on the Right Hand Side (RHS) of the := must either be a number or another memory reference. The following example shows legal RHS expressions:

```
?template b64 is [0, 64] is 0u end
?3^4:b64 := 0
?3^4:b64 := 3^4.8       -- same as "3^4:b64 := 3^4.8:b64"
```

are both legal. The first will zero out the first eight bytes of 3^4 and the second example will move the eight bytes starting at offset 8 down to the eight bytes starting at 0.

109 'ILLEGAL OPERAND TO +'
110 'ILLEGAL OPERAND TO UNARY -'
111 'ILLEGAL OPERAND TO BINARY -'
112 'ILLEGAL OPERAND TO *'
113 'ILLEGAL OPERAND TO /'

The arithmetic operators supported by the debugger will only operate on integers or values from memory. To use a value from memory in an expression a memory reference should be used. For example:

```
?template extract is [0, 16] is 0u end
?5 + 3^4:extract
```

will add the integer 5 and the first 16 bits of the segment 3^4. Note that if an address is used without a template, the default template used by the debugger is "EXTRACT":

```
?5 + 31^0f          -- same as "5 + 31^0f:EXTRACT"
```

114 'ATTEMPT TO DIVIDE BY ZERO'

115 'ILLEGAL OPERAND TO REM'
116 'ILLEGAL OPERAND TO MOD'
117 'ILLEGAL OPERAND TO **'

See the discussion of error 113.

118 'ILLEGAL PHYSICAL ADDRESS EXPRESSION'
119 'ILLEGAL INTERCONNECT ADDRESS EXPRESSION'

When keying in a physical address or an interconnect address, the expression to the right of the exclamation point(s) must evaluate to a non-negative number.

120 'OBJECT TABLE DIRECTORY ADDRESS IS NOT KNOWN'

The debugger requires that the object table directory be in a known, consistent state before logical addressing is permitted. The "DEBUG" command will cause the debugger to check the object table directory for consistency (i.e., the "DEBUG" command will make the object table directory address known). See also errors 129 - 139.

121 'BAD VALUE FOR DIRECTORY OR SEGMENT INDEX IN AD'

One (or both) of the two access descriptor coordinates is not a positive number. For example:

```
?3^(-1)
```

would cause this error.

122 'ACCESS DESCRIPTOR INDEX "<number>" TOO LARGE'

The two coordinates of an access descriptor must be in the range 0..0FFF (base 16). If either of the coordinates are larger than 0FFF this error will be displayed.

123 'CANNOT GET AN AD FROM "<address>", IT IS A DATA SEGMENT'

This error occurs when specifying the "OF process" clause of a breakpoint. The rule for evaluating the process address is: If the address is the address of a process_as use the address. If it is not the address of a process_as then use the access descriptor at the given address. If that access descriptor is not the address of a process_as, then issue error 123. For example (assume 8^1 is a process_as and 8^2 is not):

```
?4^3.9:ad          -- AD number 9 in 4^3 is for process_as
8^1
?4^3.8:ad
8^2
BA 3^1.1A43 of 8^1
BA 3^1.1A43 of 4^3.9   -- same as "of 8^1"
BA 3^1.1A43 of 4^3.8   -- err 123
```

124 'BAD LEFT OPERAND TO DOT OPERATOR; EXPECTING AN AD'
125 'ILLEGAL EXPRESSION USED AS OFFSET'

A legal logical address must be of the form "i^j.k". I, j, and k may be expressions, however the expressions must evaluate to positive numbers. I and j are restricted as described above for error 122. Error 124 is given if the value to the left of the dot, "i^j" in the example, is not an access descriptor (AD). Error 125 is issued if the expression used for "k", the value to the right of the dot, does not evaluate to a positive number or zero.

126 'OFFSET EXPRESSION IS TOO LARGE OR NEGATIVE'

A legal address must be of the form "i^j.k". This error is displayed if (k > 0FFFF).

127 'ADDRESS TO THE LEFT OF THE DOT IS ILLEGAL'

This error occurs when the right operand to the dot is a number. In this case, the left operand must be a logical address (i.e., it may not be a physical address or an interconnect address) and the logical address may not already have an offset.

128 'OPERAND TO 'SD MUST BE A LOGICAL ADDRESS'

The Segment Descriptor attribute, "'SD", only makes sense for addresses which can have segment descriptors. Recall that the definition of 'SD is:

```
    i^j'SD  is  2^i.(16*j):descr
```

Since the second entry in the object table directory is the object table entry for the directory itself, 2^i is therefore an object table and 2^i.(16*j) is the logical address of the object table entry for segment i^j. The 'SD attribute only makes sense for logical addresses. Error 128 is given if the left operand to "'SD" is not a logical address.

129 'BAD OBJECT TABLE DIRECTORY: NO I/O, NO BREAKPOINTS'

In response to the "DEBUG" command, the debugger performs a collection of consistency checks on the object table directory. If these checks fail then, although logical addressing is supported, breakpoints and I/O are not supported.

Experience has shown that the primary cause of this error is giving the "DEBUG" command after INITing, but before LOADing memory. INITing memory clears the entire memory array, which does indeed leave the object table directory in an inconsistent state.

130 'BAD PROCESSOR OBJECT TABLE: NO I/O, NO BREAKPOINTS'

The "Debug" command causes the debugger to try to find a GDP processor object which will have the access descriptors for the I/O and Breakpoint communication segments. If, while trying to find the processor, it is discovered that the processor object table (i.e., the segment 2^1) is not good, this error is displayed. This error message follows one of messages 276 - 280, which indicates what is wrong with the object table entry for 2^1. Notice that although the processor object table is bad, logical addressing is supported.

131 'NO PROCESSORS IN TABLE 2^1: NO BREAKPOINTS, NO I/O'

See also error 130. Object table 2^1 should contain the object table entries of the processor objects. If the length of table 2^1 is so short that it cannot contain any processor objects, this message is displayed. When the debugger searches the processor object table, it is looking for a GDP processor object (base type = AS, system type = processor access segment). If the debugger cannot find an object table entry for a GDP processor object in the processor object table, error 131 is displayed.

132 'PROCESSOR_AS TOO SMALL: NO BREAKPOINTS, NO I/O'

See also errors 130 and 131. Having found the lowest numbered GDP processor object in the system, the debugger checks to make sure it is long enough to contain the two access descriptors for the breakpoint and I/O communication segments. If not, error 132 is issued.

133 'BAD I/O SEGMENT/ACCESS DESCRIPTOR: NO I/O'

Before the debugger begins polling for I/O (or Breakpoints), it checks the access descriptor of the communication segment to see that it references a valid object. If either the access descriptor or the object table entry is not good, one of the messages 276 - 280 will be displayed indicating the reason, followed by this message (or message 137) to indicate the bad segment.

135 'I/O SEGMENT HAS BAD BASE/SYSTEM TYPE: NO I/O'

The debugger expects the I/O (and Breakpoint) communication segment(s) to have a system type of "GENERIC" and a base type of DATA SEGMENT. Error message 135 (or 139) will be dispayed if the segment(s) does not have the correct type.

136 'PROCESSOR_AS TOO SMALL: NO BREAKPOINTS'

It is possible for the processor access segment to be large enough to contain an I/O communication segment access descriptor, but not large enough to contain a breakpoint communication segment access descriptor. If this happens, error 136 is displayed. See also error 132.

137 'BAD BREAKPOINT SEGMENT/ACCESS DESCRIPTOR: NO BREAKPOINTS'

See Error message 133.

139 'BREAK SEGMENT HAS BAD BASE/SYSTEM TYPE: NO BREAKPOINTS'

See Error message 135.

140 'ILLEGAL VALUE USED AS ADDRESS'

The SAVE and RESTORE commands require physical addresses to be used as parameters:

```
?save !0 to !100 to :f1:myfile
?save !0 len 101 to :f1:myfile   -- these two are ok

?restore :f1:myfile to !0 to !100
?restore :f1:myfile to !0 len 101  -- these two are ok
```

141 'NEW VALUE MUST BE A NUMBER'
142 'ILLEGAL VALUE, MUST BE IN [2 .. 16]'

Both the BASE and SUFFIX commands require that the new BASE (or SUFFIX) setting be a positive number in the range 2 .. 16. Although it is an exception, the only place in the debugger where the current SUFFIX setting does not apply is when the user is inputting a radix. In these cases the radix used to interpret the number is always 10. For example:

```
?SUFFIX 16
?BASE 16
?SUFFIX 10
```

If ALL numbers were assumed to have the radix given by the most recent SUFFIX command, the "BASE 16" command given above would be illegal (16#16# is 10#22#, which would cause error 142) and the "SUFFIX 10" command would not change the suffix, since 16#10# is 10#16#.

143 'TEMPLATE RECURSION (OR NESTING) IS TOO DEEP', <CR><LF>
'WAS JUST ABOUT TO EXPAND TEMPLATE: "<name>"', <CR><LF>
' IN "<name_i>"' ...

While evaluating a template application, the template recursion or nesting became too deep for the debugger to handle. The limit is 10. The debugger announces the error and gives the name of the template it would have begun to apply to memory if the error had not been caught (<name>). The debugger then unwinds the stack, giving the name of the template that was being expanded at that point (<name_i>). The first <name_i> to be displayed will be the template which used <name>.

144 'VALUE "<number>" IS TOO LARGE'

Expressions are permitted in a variety of places in a template. If one of these expressions evaluates to a number that is larger than 16#FFFF#, this error is displayed. The possible places where this can happen is the byte_start, bit_start, bit_length, and repetition_count.

145 'MAGNITUDE "<number>" OF NEGATIVE NUMBER IS TOO LARGE'

This is the counter part of message 144, if the expression is negative.

146 'ILLEGAL VALUE IN EXPRESSION'

If during the evaluation of the expression the result is an illegal value, this error occurs.

147 'BIT LENGTH IS NEGATIVE, ONE USED'

In a template application, if the bit string is described with an expression that evaluates to a negative number, this error is issued. The template application will continue, using a bit length of one.

148 'REPEAT COUNT IS NEGATIVE, ONE USED'

See error 147.

149 'BIT STRING FOR U OR S SPEC IS LARGER THAN 32 BITS

If one of the display specifications for a template field is either nU or nS, the debugger will display the extracted bit string as either an Unsigned or Signed number. However, the debugger requires that the extracted bit string be no longer than 32 bits. For example:

```
TEMPLATE foo is [0,33] is 0u end
```

would cause the error, when applied to memory.

150 'BAD BINARY ENUMERATION VALUE'

To display an enumeration in a template field, the debugger extracts the requested bit string from memory and then compares the value extracted with the value associated with each enumeration constant. If none of the enumeration values match the value extracted from memory, error 150 is issued. For example:

```
?TEMPLATE foo is [0,2] is (zero, one, two) END
?!0:foo := 3
?!0:foo
```

the last line would cause the error to be issued.

151 'DEFAULT <DISPLAY_LIST> TEMPLATE, "<name>", IS NOT DEFINED'
 <DISPLAY_LIST> => "DISPLAY_LIST" ¦ null string

If the word "DISPLAY_LIST" is not part of the message, then the debugger was trying to find a default for an expression typed into the debugger directly:

```
?!0 := 1        -- default (for !0) is "B8"
?5 + !0         -- default (for !0) is "EXTRACT"
?!0             -- default is "DS"
?1^1            -- default is "AS" (assume 1^1 is an access
                -- segment)
```

If the word "DISPLAY_LIST" is part of the message, the debugger was trying to find a default template to use inside a template, depending on the bit string descriptor:

```
TEMPLATE foo is
  [0, 8];       -- default display_list template is "BS"
  @1;           -- default display_list template is "AD"
  baz;          -- default display_list template is "BAZ"
END;
```

If the debugger tries to use one of these defaults and it is not defined, error message 151 is displayed.

152 'NO DEFINITION FOR "<name>", EXPECTING A TEMPLATE'
153 'TRYING TO USE DEFAULT, "<name>", BUT IT IS NOT A TEMPLATE'

Just before starting a template application, the debugger looks up the name being used as the template. If the name is not in the DST the debugger issues error 152. If the name is in the DST but not defined to be a template, error 153 is given. For example:

```
?remove foo
?5^1:foo       -- This will result in error 152

?remove b8         -- remove default memory modification
                   -- template
?b8:integer := 0   -- redefine "B8" to be an integer
?!0 := 1           -- This will result in error 153 for "B8"
```

154 'VARIANT NESTING IS TOO DEEP'

When the debugger is reading a template definition, it stores the information relating to variant records in a large internal buffer. At any one time, the amount of space being used in the buffer is a complex function of the number of open (nested) variants, the number of when clauses, and the number and complexity of alternatives in each when clause. For any but the most extreme cases, the buffer should be ample.

To get rid of this error, break up single when clauses with a long list of alternatives into multiple when clauses that do the same thing.

If it is suspected that the error is a result of nesting variants too deep, one solution is to put the inner variants in a different template. For example:

```
Template foo is
  CASE [0,8] is
    when 1 =>
      CASE [1,7] is
         ...              Make this a template
      END case;
         ...
   END case
 END
```

The inner part of the nesting can be made into a template, which is then used by the above template:

```
Template inner is
   CASE [1,7] is
     ...
   END case;
 END
```

the template foo now becomes:

```
Template foo is
  CASE [0,8] is
    When 1 =>
      [0, inner];
        ...
  END case;
END
```

155 'TOO MANY <CHOICES> IN ONE WHEN CLAUSE'

When the debugger is reading in a when clause in the variant part of a template, it has an internal buffer to store the representation of the alternatives for the when clause. The maximum number of choices is approximately 120 if each choice is a single number less than 128. The minimum number of choices is approximately 20 if each choice is a range, i..j, where both i and j are too large to represent in 16 bits. If each alternative is a number too large for 16 bits, about 40 choices would be permitted.

156 '"OTHERS" MUST BE LAST WHEN CLAUSE IN VARIANT'

157 'NO TEMPLATE SPECIFIED AND THE DEFAULT, "<name>", IS NOT <def_tem>'
<def_tem> --> "DEFINED" ¦ "A TEMPLATE"

If an address is input to the debugger and no template is associated with the address, the debugger selects a default name and tries to use that name as a template. If that name is either not defined or not a template, error 157 is displayed. For logical addresses, the name selection algorithm is actually two level, such that if the first choice is not defined or not a template a second choice (either AS or DS) is made. For example (assume 1^1 is a processor_as):

```
?remove processor_as        -- The first default
?remove as                  -- the second default
?1^1
PROCESSOR_AS
ERR 157: NO TEMPLATE SPECIFIED AND THE DEFAULT, "AS", IS NOT DEFINED
```

The only templates for which this error is issued is AS and DS.

159 'OPERAND TO LEFT OF ":" IS NOT AN ADDRESS'

The binary operator ":" is used to say, "apply the template on the right to the address on the left." If the operand on the left of the colon is not an address, error 159 is issued.

160 '"<name>" IS NOT DEFINED TO BE A TEMPLATE'

Only templates may be applied to memory. for example:

```
?remove b16
?b16: integer := 0
?2^3:b16
```

Would cause error 160. Error 160 is issued in response to a template application typed in directly. This is (slightly) different than the case presented for error 153.

161 '"<name>" IS NOT DEFINED, EXPECTING A TEMPLATE'

Keying in a template application will cause the debugger to look up the template name to get its definition from the DST. If no definition is found for the name, error 161 is issued. See also error 152.

162 'INTER-PROCESSOR MESSAGE MUST BE AN ORDINAL'

When sending an IPC message, the message must be a non negative number.

163 'INTER-PROCESSOR MESSAGE TOO LARGE'

The range of values that are acceptable to the debugger for an inter-process message is 0..16#FFFF#.

164 'PROCESSOR NUMBER MUST BE AN ORDINAL EXPRESSION'

The only values allowed for processor numbers are non-negative numbers.

165 'PROCESSOR NUMBER TOO LARGE'

This message is displayed if the processor number is larger than 16#FF#.

166 'PROCESSOR NUMBER MUST BE ORDINAL'

The number for a processor in the START command must be a non-negative number.

167 'CANNOT FIND A DEFAULT PROCESSOR'

If the processor number is absent from the start command, then the debugger will, by default, start the lowest numbered GDP in the system. If there is no GDP in the system, error 167 will be displayed.

If any of the messages 129-131 were printed during the "DEBUG" command, an attempt to start the default processor will get error 167.

168 'TOP OF MEMORY MUST BE A PHYSICAL ADDRESS'

The value given to the INIT SYSTEM command should be the physical address of the last byte in memory. The debugger will use the 256 bytes just below the top of memory for operation of the 43203.

169 'ADDRESS "<number>" TOO LARGE; TOP OF MEMORY IS AT MOST 0FFFFFF'

The physical address space of the 432 is 24 megabytes, starting at 0. Therefore the highest address possible is 16#FFFFFF#.

170 'ADDRESS "<number>" TOO SMALL; MUST LEAVE 256 BYTES FOR DEBUGGER'

The debugger uses the 43203 to interface between the Series III and the 432. The debugger requires 256 bytes of 432 memory in order to make the 43203 work correctly. For example:

```
    ?INIT SYSTEM !255    -- 10#255# is too small, causes error 170
    ?INIT SYSTEM !256    -- 10#256# is ok
```

The INIT SYSTEM command saves the first 256 bytes of 432 memory, moves the 43203 control window to the last 256 bytes of memory and then restores the first 256 bytes of memory. Therefore, the physical address given on the INIT SYSTEM command should be at least !512, or some spurious error messages will be given.

172 'THE FIELD "<name>" MAY NOT BE USED BEFORE A DOT

The only template fields which may be invaded via the dot notation are those fields defined with a bit string descriptor of the form "@<expr>".

173 '"<name>" IS NOT DEFINED, USED IN <where>'
174 '"<name>" IS NOT A TEMPLATE, USED IN <where>'

```
<where> =>  EXPR BEFORE LEFTMOST DOT
          | EXPR BEFORE TRAILING "^"
          | 'ACCESS' CLAUSE
          | MOST RECENT REFERENCE
```

These two messages occur as a result of the debugger trying to use <name> as a template. These two errors are reporting specific instances of errors 152 and 153, respectively. The following example shows where each of the errors would occur:

```
?remove root
?q is 3^4:root
?q.lson                  -- ERR 173:  EXPR BEFORE LEFTMOST DOT
?q^                      -- ERR 173:  EXPR BEFORE TRAILING "^"
?root:integer := 0
?q.lson                  -- ERR 174:  EXPR BEFORE LEFTMOST DOT
?q^                      -- ERR 174:  EXPR BEFORE TRAILING "^"

?Template root is
?? lson: @0 access node;
?? rson: @1 access node;
??end
?q is 3^4:root
?remove node
?q.lson.                 -- ERR 173:  'ACCESS' CLAUSE
?node: integer := 0
?q.lson.                 -- ERR 174:  'ACCESS' CLAUSE
?q

lson: 3^1    -- applying
rson: 3^2    --  root to 3^4
?remove root
?.                       -- ERR 173:  MOST RECENT REFERENCE
```

If, instead of the "remove root" command (2nd to last in example) there had been a:

```
?remove root
?root: integer :=  0
?.                      -- ERR 174:  MOST RECENT REFERENCE
```

175 '"<name1>" IS NOT A FIELD OF TEMPLATE "<name2>"'

Using the dot notation, the user indicates a specific field of a template. However, when the field names of that template are searched, the field name is not found. <Name2> is the name of the template that should contain <name1>:

```
?template foo is
?? baz: [0,8] is 0u,/;
?? gorn:[1,8] is 0u;
??end
?q is 3^5:foo
?q.alpha                -- causes error 175
```

The message displayed for the example would be "ALPHA IS NOT A FIELD OF TEMPLATE FOO".

176 'DST IS EMPTY; MAY NOT BEGIN EXPRESSION WITH A DOT'

An expression may begin with a dot if a previous expression either ended in a dot or could have ended in a dot. However, the OUT command resets debugger status so that the DST is cleared and all previous expressions are forgotten. An expression beginning with a dot after an OUT command and before a reference will get error 176.

177 'NO PATH TO PRINT: DST IS EMPTY'

This error is given in response to a PATH command when there is no path. This error will be given right after an OUT command is used.

178 'INVALID EXPESSION PRECEEDING ".ALL"'

The debugger will only permit a memory reference to precede a ".ALL" request. Any other expression will result in this error.

179 'DST IS EMPTY; NO WHERE TO GO BACK FROM'

This message is displayed in response to a BACK command when the DST is empty.

180 'CANNOT GO BACK ANY FURTHER'

The debugger will not permit the BACK operator to clear the DST. Therefore, when there is only one segment in the "path", then a BACK command will cause this error:

```
?path
(6^3:NODE).RSON(4^5)
?back
?path
(6^3:NODE)
?back           -- will result in error 180
```

181 'EXPRESSION IS TOO COMPLEX'

There are certain places in a templates where expressions are permitted. These expressions must be buffered and saved away in the debugger's symbol table, since they are evaluated when the template is applied to memory. If any single expression is too complex (i.e., occupies too much space to fit in an internal buffer), error 181 is issued.

182 'WORKING STACK SPACE OVERFLOW'
184 'WORKING STACK OVERFLOW'

The debugger has a working stack where a most expression evaluation takes place. If the stack overflows one of these two messages will be displayed. At this point the current command will be aborted and the stack will be reset to be empty.

185 'THE IDENTIFIER "<name>" IS NOT INITIALIZED; ONE USED'

During the application of a template to memory the debugger has tried to evaluate an expression involving <name>. Although <name> is defined to be an integer, it was not given a value. The error can be fixed by simply removing <name>, redefining it with the desired value, and then reapplying the template to memory. For example:

```
?Template ds is
??[by_p:bi_p+1,16] is <row_count>[<8>0u:9,/];
??end
?row_count:integer
?2^2:ds               -- will cause error 185
?remove row_count
?row_count:integer := 4
?2^2:ds               -- will work ok
```

186 '"<name>" MAY NOT BE USED AS A NUMBER'

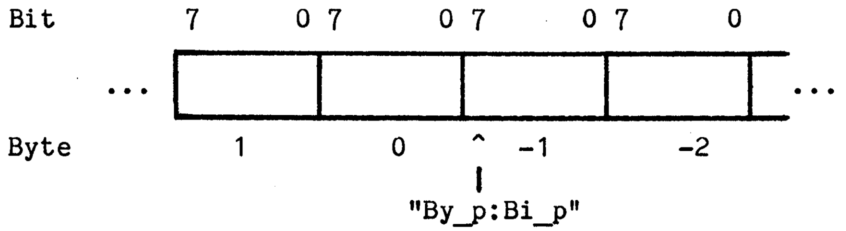
The only names that are permitted in expressions inside of templates are names having the type INTEGER. An attempt to use any other type of name will result in error 186.

187 '"<name>" IS NOT DEFINED'

While trying to evaluate an expression, during an application of a template to memory, the debugger looked up a name and discovered it was not defined, see also error 185.

188 'NORMALIZED BYTE:BIT OFFSET IS NEGATIVE (BYTE)'
189 'NORMALIZED BYTE:BIT OFFSET IS NEGATIVE (BIT)'
190 'BOTH BYTE AND BIT OFFSETS ARE NEGATIVE'

The following model is used when applying templates to memory:



The "bit steam" that the template is applied against starts at bit 0 of byte 0. The picture also shows the where the Byte and Bit pointers are initially: By_p is -1, Bi_p is 7. When the debugger is evaluating a bit string descriptor, it first evaluates both the byte and bit offset, and then normalizes them; trying to make both numbers non-negative and the bit offset less than 8. If it cannot make both numbers non-negative, one of the above errors will be issued, depending on which value, bit offset or byte offset, was initially negative. For example:

```
?Template ds is
?? [by_p:bi_p, 16] is <4>[<8>0u:9,/];
??end
?2^2:ds
```

will cause error 188, since "by_p:bi_p" is "-1:7", and this cannot be normalized. However by rewriting the bit string descriptor to "[by_p:bi_p+1, 16]", the error is fixed:

```
[by_p:bi_p+1, 16]     =     [0:0, 16]     initially.
```

191 '"<name>" IS ALREADY DEFINED'

When defining a memory REFERENCE, if the name to the left of the "IS" is already defined, error 191 is given. For example:

```
?foo: integer :=0
?foo is 5^4
```

Would result in error 191.

This error also occurs during template definition, if either the template name has already been defined or if an attempt is made to define two fields in the same template with the same name.

192 'ILLEGAL DEFINING EXPRESSION: MUST BE A TEMPLATE APPLICATION'

This error is issued when a memory reference is being defined, but the expression after the "IS" is not a template application (i.e., a reference).

194 '"<name1>" IS NOT <def_tem>. THEREFORE "<name2>" IS INVALID'
<def_tem> --> "DEFINED" ¦ "A TEMPLATE"

This error occurs during the application of a memory reference. If the template name used to define the reference is either not defined or not a template error 194 will be issued. See also errors 152 and 153. For Example:

```
?foo is 5^3:baz
?remove baz
?foo          -- "DEFINED"    name1: BAZ, name2: FOO
?baz:integer := 0
?foo          -- "A TEMPLATE" name1: BAZ, name2: FOO
```

195 'ADDRESS "!<number1>" TOO LARGE; !<number2> IS THE TOP OF MEMORY'

This error is issued when a physical address is larger than the last value given for Top Of Memory. <number2> is the current value for Top Of Memory being used by the debugger, displayed in the current output radix.

196 'BAD ADDRESS: "<log_address>". OFFSET IS TOO LARGE.'

If the offset portion of a logical address is too large, for example if it extends beyond the end of the segment, this message is given. Remember that if the segment is an access segment, then the the offset stands for access descriptor index NOT byte index. For example, assume that 3^4 is an access segment containing 8 access descriptors (3^4.0 through 3^4.7):

```
?3^4.7      -- ok
?3^4.8      -- error 196
```

197 '"<log_address>" IS NOT AN INSTRUCTION OBJECT ACCESS DESCRIPTOR'

The <log_address> is not the address for an instruction object. Instruction objects are required for BA breakpoints.

198 'THE INSTRUCTION OFFSET OF THE ADDRESS "<log_address>" IS TOO LARGE'

Before trying to set a BA breakpoint, the debugger verifies that the bit offset given in the <log_address> is actually inside the instruction data segment. If the bit offset is too large, then error 198 is given.

199 'THE DIGIT "<digit>" IS TOO LARGE FOR INPUT RADIX: <number_10>'

The debugger recognizes any string beginning with a digit and containing digits and/or the letters "A".."F" as being a number. Once it has recognized the characters as a number, the debugger attempts to translate the ascii string into binary, using the current input radix as set by the most recent SUFFIX command. If one of the characters is not a valid digit in the current input radix, then that digit (<digit>) and the current input radix (<number_10>) are displayed. The current input radix is always displayed in base 10.

```
?suffix 8
?128          -- error 199, bad digit is 8, input radix is 8.
```

200 'RADIX OF NUMBER MUST BE BETWEEN 2 AND 16

The user may override the current default input radix by using the notation: "r#n#", where r is a decimal number used as the input radix for the number, n. If r is not in the range 2..16 error 200 is issued.

201 '"<name>" IS IN SYMBOL TABLE, BUT IS NOT INITIALIZED'

<Name> is an integer, but was not given an initial value. It may not be used in an expression.

202 'TEMPLATES (I.E., "<name>") MAY NOT BE USED AS EXPRESSION PRIMARIES'

This error happens when the <name> of a template is used where a number should be used.

203 '"<name>" MAY NOT BE AN IDENTIFIER IN AN EXPRESSION'

The debugger supports a variety of types for names. Integers, references, and template fields (under certain circumstances) may be used in expressions. Breakpoints may not.

204 'EXPRESSION HAS TOO MANY TERMS'

When the debugger is reading in a template definition it keeps an internal buffer for expressions. These expressions can occur as the <Byte_start>, <Bit_start>, <Bit_length> and <Repetition> parts of a template field. The "complexity" of each expression is limited by the size of the buffer. An expression does not have too many terms if "4*o + 2*n < 240" is true, where o is the number of operands and n is the number of operators.

205 'ILLEGAL OPERAND IN TEMPLATE EXPRESSION'

206 'THE <by_bi> "<number>" MAY NOT BE GREATER THAN OFFFF'
<by_bi> --> "BYTE START" ¦ "BIT START"

This is a sanity check performed by the debugger at template definition time. The byte start and bit start values must fit into 16 bits, if not, this error is given.

207 'THE AD INDEX "<number>" MAY NOT BE LARGER THAN OFFF'

This is another sanity check performed at template definition time. It is impossible to have an AD index larger than OFFF, since segments can be at most OFFFF in length and each access descriptor is four bytes long.

209 'MAY NOT FOLLOW ACCESS ATTRIBUTE WITH A DOT'

It is not legal to have a template field of the form:

```
?template foo is
?? @0 access foo.[0,8];
??end
```

The access clause must always come after the bit string descriptor and may only appear if the bit string descriptor is an AD index or an AD index preceded by a chain of "@number.".

210 'REPETITION COUNT EXPRESSION IS TOO COMPLEX'

This error is the same as error 204, except that it is explicitly for the repetition part of a display field.

211 'NESTING TOO DEEP IN DISPLAY PORTION OF FIELD'

The maximum nesting depth when defining a display specification is 4. For example:

```
?template foo is
??[By_p:bi_p+1, 8] is <3>[<2>[<2>[<2>[0u:9]]], /];       -- ok
??[By_p:bi_p+1, 8] is <3>[<2>[<2>[<2>[<2>[0u:9]]]], /]; -- err 211
??end
```

212 'REPEAT COUNT IS TOO BIG'

The largest permissable repeat count is 16#FFFF#. Anything larger will cause error 212 and the repeat count to be ignored (i.e., one will be used).

213 '"<nU>" HAS AN ILLEGAL RADIX VALUE'

The <nU> is an integer conversion display specification of the form "nU" or "nS". If the value of n is not 0 or in the range 2..16 then this error is given. For example:

```
?template foo is
?? [0,8] is 0u, 3u, 8s, 13u, 16s;   -- these are fine
?? [0,8] is 1u, 17u, 212s;          -- these cause error 213
??end
```

214 '"<number>" IS TOO LARGE TO BE A DISPLAY WIDTH.'

The largest permissable display width is 64. Attempting to use a larger value will cause this error, and 16 will be used.

216 'VALUE OF "<name>" IS OUT OF ORDER, <number> WILL BE USED'

For enumerations, the debugger requires that the values of the enumeration constants be given in order. If they are not, this error is given, indicating which name is out of order. The debugger will replace the value specified in the enumeration with the next available value:

```
?Template foo is
?? [0,8] is (1=>one, 8=>eight, 7=>seven, 64=>sixty_four);
??end
```

This would cause an error when "7=>seven" is read in. The value for seven would be 9, the next available value.

217 'ENUMERATIONS MUST BE EITHER ALL IMPLICIT OR ALL EXPLICIT'

The debugger does not support mixing enumeration constants by giving some explicit values and some implicit values. for example:

```
?Template foo is
?? [0,8] is (alpha, beta, gamma, tau, epsilon); -- ok
?? [1,3] is (0=>ha, 1=>ee, 2=>yut, 3=>kah);     -- ok
?? [2,3] is (1=>uno, 2=>dos, quatro, cinquo);   -- err 217
?? end
```

218 'ENUMERATION VALUE "<number>" IS TOO LARGE; MUST FIT IN 16 BITS'

The debugger restricts enumeration values such that they must fit into 16 bits.

219 'DISCRIMINANT IS OUT OF RANGE AND NO OTHERWISE CLAUSE'

If, while applying a template to memory, the debugger discovers a variant discriminant for which there is no WHEN clause, including no WHEN OTHERS clause, this error is issued:

```
?template foo is
??case [0,8] is      -- first byte is the discriminant
??  when 0 =>
??      foo: b16;
??  when 1 =>
??      baz: b16;
??end case;
??end
```

If foo is applied against memory where the first byte has any value except 0 or 1, error 219 is given.

221 'INCLUDES NESTED TOO DEEP'

The maximum nesting of includes is 4.

222 'BAD FILE SPECIFICATION'

For the DEBUG, LOAD, SAVE, RESTORE, INCLUDE, and LOG commands, this error will be issued if the file specification is not a valid Series III file name.

223 'NO LOG FILE ACTIVE. LOG COMMAND IGNORED.'

Once a log file has been established via a "LOG file" command, the debugger permits logging to be directed just to the CRT or just to the log file via the ">CRT" and ">LOG" commands. If a ">LOG" command is given and there has been no "LOG file" command, then error 223 is given. For example:

```
?>log         -- error 223
?log :f1:log
?>crt         -- ok
?>log         -- ok
```

224 '432 IO REQUESTED ON ILLEGAL DEVICE'
225 'ILLEGAL IO FUNCTION REQUESTED BY 432'
226 '432 REQUESTED INPUT FROM ILLEGAL DEVICE'
227 '432 REQUESTED OUTPUT ON ILLEGAL DEVICE'
228 'LENGTH OF 432 IO REQUEST > 132'

After any of these error messages, the values in the first 8 bytes of the I/O communication area are printed as follows:

```
'FUNCTION_REQUESTED:' <function requested>
'FILE_AFTN:         ' <file aftn>
'FUNCTION_COMPLETED:' <function complete>
'LENGTH:            ' <length of io request>
```

During normal operation, the debugger is polling three different places to see if any activity is taking place: Breakpoints, I/O, and the keyboard. If an I/O request is issued by the 432, then the debugger checks all of the parameters of that request to insure that they are valid. If not, one of the above messages is issued, along with the four 16-bit parameters.

229 'ILLEGAL CHARACTER'

If an illegal character is detected by the debugger, this message is displayed. The illegal characters to the debugger are (in hex): 0, 1, 5, 6, 7, 8, 9, 0B, 0C, 0E, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1D, 1D, 1E, 1F, 60, 7E, 7F. The control characters 2(^B), 3(^C), 4(^D), 0A(LF) 0D(CR), 0F(^O), 18(^X), and 7F(RUBOUT) have special meaning at the user interface. Of particular importance is ^D, which causes the 8086 debugger to be entered.

230 'TOKEN TOO LARGE'

The maximum size token permitted by the debugger is 255 characters.

231 'ILLEGAL END-OF-FILE ENCOUNTERED'

If the debugger encounters an end of file in an INCLUDE file, this error is issued.

232 'ILLEGAL SYNTAX'

The debugger has an internally-encoded grammar of the debugger command language and requires commands to conform to that grammar. The first input token that caused the error is indicated with an up-arrow underneath it (^). For example:

```
?3+
```

would result in a syntax error, with the ^ pointing just after the "+". In this case the illegal token is the end-of-line, since the grammar requires a right operand to "+". Another example:

```
?template break is
```

will cause error 232 with an the ^ pointing to break; break is a keyword to the debugger and may only be used as a keyword (e.g., dir break).

```
233  <CR><LF>' DEB432 I/O ERROR - ',<CR><LF>
    '   FILE:        ' <filename> '<CR><LF>
    '   OPERATION: <  'CLOSING A FILE'
                   ¦ 'ATTEMPTING DELETION'
                   ¦ 'SETTING CONTROL-C TRAP'
                   ¦ 'SETTING SINGLE CHARACTER MODE'
                   ¦ 'TRYING TO SEEK'
                   ¦ 'TRYING TO SEEK RELATIVE'
                   ¦ 'GET_POSITION FOR FILE W/ NO SEEK SUPPORTED'
                   ¦ 'TRYING TO EMPTY'
                   ¦ 'TRYING TO READ IN AN OVERLAY'
                   ¦ 'INITIALIZING THE CONSOLE FOR OUTPUT'
                   ¦ 'OPENING FOR LOG'
                   ¦ 'INITIALIZING THE CONSOLE'
                   ¦ 'RE-OPENING OUTER INCLUDE'
                   ¦ 'OPENING FOR INCLUDE'
                   ¦ 'OPENING AN LIF'><CR><LF>
    '   ERROR:       ' <UDI error message><CR><LF><CR><LF>
```

All debugger I/O errors are reported via this error message. There are three variable parts. The first is the name of the file/device that is causing the error. This is preceded by the word "FILE:". The second part of the message is a description of the debugger operation that was being attempted when the error happened. this is preceded by the word "OPERATION:". The last piece of information is the standard UDI diagnostic for this particular I/O error. This message is preceded by the word "ERROR:". For more information on the error, see the Series III user's manual.

234 'CURRENT PROCESS IS NOT AT A BREAKPOINT'

This message is issued when a resume command, without a process, is given, but the default process (i.e., CP, the current process) is not currently in the process set of broken processes.

236 'PROCESS IS NOT AT A BREAKPOINT'

This error is the result of trying to select or resume an explicit process that is not in the set. The "examine" command will display all of the processes currently in the process set.

237 'NO CURRENT CONTEXT'

This error is given in response to any of the current context stack operators: UP, DOWN, TOP, or BOTM, when there is no current context (i.e., no call stack to move around on).

238 'ALREADY AT TOP OF CALL STACK'
239 'ALREADY AT BOTTOM OF CALL STACK'

Giving either a TOP or BOTM command when the current context is already located at the top or bottom of the call stack causes these errors.

240 'REQUESTED STACK DEPTH > 2**16. ENTIRE STACK WILL BE DISPLAYED'

This error is a result of a "STACK n" command where n > 16#FFFF#. The entire stack is displayed, even if the current context is not at the top.

241 'TRIED TO SET OR ACTIVATE TOO MANY BREAKPOINTS'

The debugger has a limit of 32 active breakpoints at any one time. Error 241 occurs when trying to set/activate the 33rd breakpoint. The corrective action is to get a directory of the breakpoints, "DIR break", noting the unstarred breakpoints, as these are the ones that are active. At least one of these breakpoints must either be deactivated or removed. Then the breakpoint can be set.

Although the debugger has a limit of 32 breakpoints, each of these breakpoints may cause more than one process to break, since processes may be sharing the same instruction object. Therefore the actual number of breakpoints that can be set is much larger than 32.

242 'BAD MEMORY: BREAKPOINT COMMUNICATION SEGMENT IS MISSING'
243 'BAD MEMORY: ILLEGAL INSTRUCTION OBJECT ADDRESS'
244 'BAD MEMORY: ILLEGAL INSTRUCTION OBJECT OFFSET'
245 'BAD MEMORY: ILLEGAL PROCESS ADDRESS'

The debugger is constantly double checking all of the values that it gets from the 432, since the 432 is still running, even though some of the processes may have breakpointed. If it finds an inconsistency, while fielding a breakpoint, one of the four messages above will be given. Error 242 indicates that the breakpoint communication segment is all of a sudden missing. Errors 243-245 indicate what value has been discovered to be bad.

246 'THERE IS ALREADY A BREAK SET AT THIS POINT'

There may be only one occurrence of a particular breakpoint definition for a particular process.

247 'WARNING: PROCESS BROKE DURING ATTEMPT TO RESUME'

To resume from a BA breakpoint that has not been deactivated or removed, the debugger replaces the illegal class code it used to set the breakpoint with the good class code, single step the process over the instruction, then sets the illegal class code back into the instruction object. If, when trying to single step the process over this instruction, the process "breaks", then this error is issued. This will "typically" happen if two or more processes are broken at the same location and are resumed at the same time. For example, assume there are three process, each in a loop calling a shared instruction object (8^0C8):

```
?share: ba 8^0c8 of all     -- assume 3 processes 8^54, 8^9C, 8^66
?start
  ...
?
SHARE: BREAK AT: 8^0C8.50 OF 8^54
?
SHARE: BREAK AT: 8^0C8.50 OF 8^9C
?
SHARE: BREAK AT: 8^0C8.50 OF 8^66
?resume all                 -- resume all three processes
?
SHARE: BREAK AT: 8^0C8.50 OF 8^54
?
ERR 247: PROCESS BROKE WHILE TRYING TO RESUME
SHARE: BREAK AT: 8^0C8.50 OF 8^9C
?
ERR 247: PROCESS BROKE WHILE TRYING TO RESUME
SHARE: BREAK AT: 8^0C8.50 OF 8^66
```

The error happens after one process makes it all the way through the cycle (8^4F in the example). The final step of that process is to rewrite the illegal class code. At this point, the other processes that have not yet single stepped over the instruction will break when they try to single step, because the instruction now has an illegal class code.

The status of the process is the same as if it had not been RESUMEd. It is in a consistent state and may be resumed at any time. The error is really just a warning and a piece of information: "the process did not get resumed."

248 'CANNOT CLEAR TRACE. PROCESS NOT AT A BREAKPOINT'

For any of the tracing breakpoints (BO, BE, BX), the process must be at a breakpoint or else the trace mode cannot be changed.

249 'INSTRUCTION SPECIFICATION MUST BE LOGICAL ADDRESS'

When setting a BA breakpoint, the address of the instruction must be a logical address, i.e., a segment with an optional bit offset.

250 'ILLEGAL BIT OFFSET FOR THIS INSTRUCTION DATA SEGMENT'

The debugger checks to see if the bit offset given when setting a BA breakpoint is possible in the instruction data segment. If not, then this error is issued.

--- ' NB: NOT THE DEFINING PROCESS'

If a breakpoint is set in an instruction object for process, p, and the breakpoint is actually hit by process q, which shares that instruction object with process p, this warning is given along with the breakpoint message when process q hits the breakpoint.

252 'ILLEGAL VALUE USED FOR <BPT>'

```
 <BPT> --> PROCESS OBJECT
         | DOMAIN OR INSTRUCTION OBJECT ADDRESS
```

The process(es) in the "OF <process_list>" clause of a breakpoint must be a logical address. The same is true for the domain or instruction object address for "BE" and "BX" breakpoints. An expression which is not a logical address in one of these places will result in this error. For example:

```
?a: integer := 0
?ba 8^05b of a      -- err 252, <BPT> -> PROCESS OBJECT
```

The error is due to "a" being an integer, not a reference.

253 '"<address>" MUST BE A LOGICAL ADDRESS'

To define a breakpoint, both the instruction address and the process access segment address must be logical addresses. For example:

```
?ba 3^4b of !210
```

is not permitted.

255 "<ad>" ' MUST BE A PROCESS OBJECT'

When defining a breakpoint, the debugger checks to make sure that the address given in the "of" clause is a process. If not, then this error is given.

256 '"<ad>" MUST BE A DOMAIN OR INSTRUCTION OBJECT ADDRESS'

When defining a BA breakpoint, the location of the breakpoint must be in an instruction object. When defining a BE or BX breakpoint, the location of the breakpoint is either an instruction object or a domain.

257 'NO CURRENT PROCESS TO USE AS DEFAULT'

If a breakpoint is defined and there are no processes currently breakpointed, the breakpoint definition must contain an "OF <process_ad>" clause. This should only happen before the first process hits a breakpoint. For example:

```
?dir cp
ERR 187: "CP" IS NOT DEFINED
?ba 5^4.122          -- uses the default, cp, as the process
                     -- and would get error 257
```

258 'PROCESS "<ad>" IS NOT AT A BREAKPOINT'

If a RESUME command specifies a particular process to resume, and the debugger has no record of that process in the set of breakpointed processes, error 258 is given.

259 'NO CURRENT PROCESS TO RESUME'

If a RESUME command is given with no specific process and there is no process breakpointed at this time, error 259 will be issued. See also error 257.

261 '"<name>" IS NOT A BREAKPOINT NAME'

This error is issued in response to either an ACTIVATE or DEACTIVATE command where <name> is not a breakpoint name. Activating or deactivating anything besides a breakpoint does not make sense.

262 'BREAKPOINT "<name>" IS ALREADY ACTIVE'
263 'BREAKPOINT "<name>" IS ALREADY INACTIVE'

Activating an already active breakpoint or deactivating an already deactivated breakpoint will result in one of these errors.

265 'FILE SAVE/RESTORE ABORTED: HEAD RECORD MISSING'
266 'FILE SAVE/RESTORE ABORTED: ILLEGAL LENGTH FOR RESTORE'
267 'FILE SAVE/RESTORE ABORTED: ILLEGAL LENGTH FOR SAVE'
268 'FILE SAVE/RESTORE FAILED: ILLEGAL RECORD TYPE'
269 'FILE SAVE/RESTORE FAILED: ILLEGAL RECORD LENGTH'
270 'FILE SAVE/RESTORE ABORTED: BAD LLA RECORD'
271 'FILE SAVE/RESTORE ABORTED: UNEXPECTED EOF IN LIF'
272 'FILE SAVE/RESTORE ABORTED: CHECKSUM ERROR IN LIF'
273 'FILE SAVE/RESTORE ABORTED: BAD DATA RECORD TYPE IN LIF'

These errors occur during a SAVE or RESTORE command. Errors 265, 268, 269, 270, 271, 272, and 273 are RESTORE errors and indicate that the file that was previously SAVEd has been corrupted.

276 '<1_2> COORD. OF "<ad>" IS BAD: IT IS BEYOND END OF O.T.'
277 '<1_2> COORD. OF "<ad>" IS BAD: O.T. ENTRY IS NOT AN O.D.'
278 '<1_2> COORD. OF "<ad>" IS BAD: O.T. ENTRY NOT VALID'
279 '<1_2> COORD. OF "<ad>" IS BAD: NO STORAGE ALLOCATED FOR O.D.'
280 '<1_2> COORD. OF "<ad>" IS BAD: ANCESTOR OF REFINEMENT IS BAD'
<1_2> --> "FIRST" ¦ "SECOND"

As the debugger interprets addresses, it checks them for validity. If an error is found while decoding an access descriptor or an object descriptor, one of the above messages is displayed.

The first word of the error message indicates which coordinate of the access descriptor is bad. For example in the access descriptor "1^2", 1 is the first coordinate and 2 is the second. If the error specifies the first coordinate, errors 277-280 refer to the object descriptor of the object table which contains the object table entry of the segment. For example, if the first coordinate of "5^2" is bad, the 5th object table entry in the object table directory is bad.

The access descriptor that was being interpretted when the error was detected is displayed, followed by the reason for the error. The following error abbreviations are used:

O.T. -- object table
O.D. -- object descriptor

Error 276 says that the bad coordinate, when multiplied by 16 (the size of object table entries), is larger than the length of the object table. Error 277 indicates that the object descriptor is not really a valid object table entry. The debugger recognizes refinement descriptors, storage descriptors, and interconnect descriptors. All others (type descriptors, header entries, and free entries) cause error 277.

Error 278 says that the "valid" bit of the selected object table entry is set to "not valid". Error 279 indicates that the "allocated" bit of the object table entry is set to "no storage allocated". Error 280 says that while trying to track down the ancestor of a refinement, the debugger ran into another refinement.

281 'RESERVED MEMORY ERROR'
282 'UNCORRECTABLE ECC ERROR'
283 'STORAGE ARRAY DATA PARITY ERROR'
284 'STORAGE ARRAY ADDRESS PARITY ERROR'
285 'MEMORY NOT PRESENT'
286 'MEMORY CONTROLLER DATA PARITY ERROR'
287 'MEMORY CONTROLLER ADDRESS PARITY ERROR'

When the debugger is given the command to LOAD memory, it checks to see if memory is initialized. This check involves stepping through memory in 16KB steps, reading and writing a byte to see that everything is OK. If a 432/670 error is encountered, then one of the messages above is issued and memory is viewed as not being initialized.

288 'ILLEGAL IPC MESSAGE'

289 'FAILED TO INITIALISE 432 SYSTEM'

In response the the INIT command, the debugger performs the following steps:

1. read the diagnostic control port (dummy read).
2. output the pattern 16#C2C2# to the diagnostic control port.
3. idle for 2 milliseconds.
4. read the diagnostic control port, looking for the pattern 16#00F7#.

If step 4 does not find 16#00F7# at the diagnostic control port, the debugger issues error message 289. For more information see the SYSTEM 432/600 SYSTEM REFERENCE MANUAL.

290 'IP FAILS TO PERFORM REQUESTED FUNCTION (TIMED OUT)'
291 'IP PROCESSOR FAULT'
292 'IP PROCESS FAULT'
293 'IP CONTEXT FAULT'

These errors are reported when the debugger tries to open a 43203 window onto memory or interconnect space and the component faults. The only 43203 instruction that the debugger executes is an "ALTER MAP AND SELECT PHYSICAL SEGMENT". After initiating execution, the debugger waits until the 43203 asserts "operation complete". If, after 15 milliseconds, the 43203 has not completed the alter map, error 290 is issued and the operation is aborted.

If the 43203 faulted, the debugger prints one of the error messages 291-293 to indicate what kind of fault and aborts the operation. Any further output from a debugger command which gets one of the above errors is suspect.

The debugger is still operational after one of the above errors. A "typical" cause of one of these errors is either a bad connection between the Series III and the 432 or someone may have turned off the 432.

294 'PROCESS_DS AD IS BAD'
295 'BAD PROCESS_AS AD DURING ATTEMPT TO CHANGE STATUS'

While trying to resume a process, the debugger must change the status of a process twice: once to "full" trace and once back to "no" trace. If the debugger discovers a bad process_as or process_ds during these operations, these messages are displayed. These are very unusual messages and indicate something is wrong in the 432 memory image.

```
296  'BAD EOD FILE, EXCEPTION '<exc>'H : '<outcome>'<CR><LF>
      <cause><CR><LF>
      <detail><CR><LF>

     <exc> ==> <4_digit_hex>
     <outcome> ==> 'LOAD FAILED'¦'LOADER WARNING'¦'LOAD ABORTED'
     <cause> ==> 'FATAL ERROR'¦
                 'UNEXPECTED EOF ON CONSOLE INPUT'¦
                 'MISSING EOD SPECIFICATION IN PREAMBLE'¦
                 'COUNT FIELD FOR ITEM SIZE > 2 IN PREAMBLE'¦
                 'EXPECTED SOG ITEM SPECIFICATION IN PREAMBLE'¦
                 'ITEM SPECIFICATION ALTERED IN PREAMBLE'¦
                 'ITEM SPECIFICATION ADDED IN PREAMBLE'¦
                 'UNEXPECTED EOF IN EOD'¦
                 'UNKNOWN ITEM ENCOUNTERED IN EOD'¦
                 'EOG HAS NO CORRESPONDING SOG: EOG EOD ASSUMED'¦
                 'SOG STACK OVERFLOW'¦
                 '"EODRESTORE CALLED WITHOUT MATCHING PREVIOUS "EODSAVE"'¦
                 'LOOKING FOR SEGMENT DESCRIPTOR ITEM'¦
                 'LOOKING FOR ADDRESS ITEM'¦
                 'LOOKING FOR DATA ITEM'¦
                 'REACHED END OF FILE WHILE LOADING DATA'¦
     <detail> ==> 'ITEM: '<ind>', TYPE:'<type>',<CR><LF>
                  [<item_info>]
     <ind> ==> <2_digit_hex>
     <type> ==> <2_digit_hex>
     <item_info> ==> 'SOURCE: '<source>', '<valid?>',
                     '<size_mode>', SIZE:'<size>[', TYPE BYTE']
     <source> ==> 'DEFAULT'¦'PREAMBLE'¦'UNKNOWN'
     <valid?> ==> 'VALID'¦'INVALID'
     <size_mode> ==> 'FIXED'¦'VARIABLE'
     <size> ==> <2_digit_hex>
```

When the debugger loads a file it must be in a specific format. The debugger checks the file as it loads it to verify that it is in the correct format. If the file does not conform this error message is issued.

If a second attempt to load the same file results in the same error message, the file is either not an output of LINK-432 or UPDATE-432 or the file has been corrupted, for example while being downloaded from a host computer.

```
297  'OVERFLOW DURING **'
298  'OVERFLOW DURING *'
```

The debugger checks for overflow while doing multiplication and exponentiation.

299 'CANNOT FIND PROCESSOR OBJECT'
300 'CANNOT FIND PROCESSOR COMMUNICATION OBJECT'

When the user sends a local IPC to a specific processor, the debugger must find the processor object and find that processor's local communication object. If either of these searches fails, then one of these messages is given. For example, if the base or system type of the communication object is incorrect, this message will be displayed.

301 'PROCESSOR COMMUNICATION OBJECT IS LOCKED'

When sending an IPC, the debugger must write into either the local or global communication object. If the debugger finds the object locked, it does not write the message into the object, it does not send the IPC signal, and it reports this error message. There are two possible reasons for this error message: (1) The debugger IPC command was executed just when a processor was either reading or writing the communication object (unlikely) or (2) a processor faulted while trying to read/write the communication object and never unlocked it.

302 'AN IPC MESSAGE IS PENDING AT THE PROCESSOR COMMUNICATION OBJECT'

When sending an IPC, the debugger checks to make sure there is not already a message in the communication object. If a message is already there, this error is reported. The most likely reason for this error is that the processor is faulted (not executing) and this is the second IPC command executed by the debugger (i.e., the message from the first IPC is pending).

311 'NUMBER, "<number>", IS TOO BIG, DOES NOT FIT INTO 32 BITS

If a number is input which, when the debugger is converting from ascii to binary, does not fit into 32 bits, this error is given.

312 'REMOVE ALL ABORTED'

This message is the second message, after a specific error message indicating which symbol could not be removed, and why. This might happen if a breakpoint cannot be removed because the OS has made the instruction data segment temporarily inaccessible. Retrying the "REMOVE ALL" may work in this latter case.

313 'TOO MANY TEMPORARY SYMBOLS ACTIVE'

The debugger does not enter symbols into the debugger symbol table until the definition is complete. This requires that the debugger buffer the definition until it is finished. If the definition of the symbol is so large (e.g., a template) that it does not fit into this buffer, this error is issued.

This is a caution message. The definition can continue, and as long as no errors happen, everything is OK. However, if an error is made after this error message, that part of the debugger's symbol table used to temporarily hold the definition is made inaccessible for the rest of this debugging session. In the worst case, this will just mean a loss of performance. The buffer is so large that only test cases have caused this error.

314 'BAD CONTEXT FOUND WHEN MOVING DOWN CALL STACK'

While trying to execute one of the stack commands, TOP, DOWN, BOTM, UP, or STACK, the debugger found that a context on the call stack is inconsistent. The value of "cc" is not changed if the command is TOP, DOWN, BOTM, or UP. In the case of a STACK command, the last context to be displayed is the last good context. The next context that would have been displayed (the "previous" of the last one displayed) is probably the bad context.

315 'UNRECOVERABLE IP FAULT'

This error will be announced after one of errors 290 - 293. It indicates that the debugger tried twice to get ALTER MAP to work, but failed. At this point, the user should re-initialize the system. The INIT SYSTEM command will enable the user to examine the contents of memory.

316 'NO ADDRESSING POSSIBLE, MUST INITIALIZE 432 HARDWARE'

If a physical or interconnect address is keyed in to the debugger, but the debugger detects that the IP board is not initialized, then the address is not evaluated. Before 432/670 system memory may be examined or modified, the user must initialize the system via either INIT or INIT SYS.

318 'IP NOT WORKING. SYSTEM UNINITIALIZED'

319 'BREAKPOINTS ARE NOT SUPPORTED FOR THIS MEMORY IMAGE'

This error occurs if the user attempts to set a breakpoint after having received one of the error messages: 129-132, 136-139 in response to the DEBUG command.

320 'TOP OF MEMORY ADDRESS IS TOO LARGE'

When trying to do an INIT SYSTEM command, the IP board reported back to the debugger that some part of the last 256 bytes of memory are not present. This means that the address given in the INIT SYSTEM command is too large.

321 'ILLEGAL EXPRESSION PRECEDING DOT, NOT A REFERENCE.

The dot operator is expecting a memory reference preceding it. This may be in the form of an explicit template application, a REFERENCE variable, or a field of either of these defined with an "@<expr>" kind of bit string descriptor. For example:

```
?template node is
??lson: @0 access node;
??lson: @1 access node;
??end;
?p is 4^3:node
?p.lson                -- the "p" before the dot is ok
?4^3:node.lson         -- the "4^3:node" before the dot is ok
?p.lson.rson           -- the "p.lson" before the dot is ok
?foo: integer := 5
?foo.                  -- the "foo" before the dot causes err 321
```

322 'ILLEGAL SYNTAX. ADDRESS MAY NOT FOLLOW "BO"'

The legal event breakpoints are "call", "fault", "inst", and "ret". These are the only options when setting an event breakpoint:

```
BO FAULT   -- is ok
BO 9^3.7A3 -- is not ok
```

323 'ILLEGAL SYNTAX. ("CALL" ¦ "FAULT" ¦ "INST" ¦ "RET") MUST FOLLOW "BO"'

See also 322. These events only make sense for an event breakpoint (i.e., Break On <Event>).

324 'LEFT OPERAND TO 'SD IS ILLEGAL

The 'SD attribute of an address returns a reference to the object table entry for the underlying segment being addressed. The only kinds of addresses that use object table entries are logical addresses. All other kinds of addresses (or any other non-address operands) used as a left operand to 'SD will give error 324.

326 '432 IS UNINITIALIZED, ADDRESSING IS NOT SUPPORTED'

If the 432 is not initialized and an address is keyed in, this error may be issued. The response is to initialized the 432 (e.g., via the INIT or INIT SYSTEM command).

327 'ILLEGAL PHYSICAL ADDRESS'

328 'BAD IP WINDOW OPERATION ON WINDOW #n. ENTRY FAULT CODE:
<reason>+'

```
<reason> => 'READ/WRITE'
          ¦ 'BUS ERROR'
          ¦ 'ACCESS RIGHTS'
          ¦ 'MEMORY OVERFLOW'
          ¦ 'ACCESS DIRECTION'
          ¦ 'POST TERMINATION'
          ¦ 'PARTIAL BLOCK OVERFLOW'
```

Before every 43203 ALTER_MAP_AND_SELECT_PHYSICAL_SEGMENT operation that the debugger performs, it checks the status of all the "windows" (i.e., 43203 map entries). If a window has faulted, then error 328 is displayed and the entry fault code bits are decoded and displayed below the error message. The debugger does not execute the ALTER_MAP it was about to attempt, but instead goes back to prompt the user for a command.

If this error occurs, the most recent debugger operation(s) on memory are suspect. For more information see the "iAPX 432, INTERFACE PROCESSOR, ARCHITECTURE REFERENCE MANUAL" (Order no. 171863-001).

The user may choose to ignore the error as a temporary glitch or to attempt to discover the cause (e.g., via the interconnect registers).

329 'FATAL IP ERROR'

Before every 43203 ALTER_PHYSICAL_MAP_AND_SELECT_DATA_SEGMENT operation the debugger performs, a check is made to see if entry map 4 (the 43203 control window) is still accessible. If it is no longer accessible, error 329 is issued. This error "typically" is the result of a bad connection between the Series III and the 432/670. Another "typical" occurrence of this error is when the power is turned off in the 432/670 system while the debugger is reading or modifying system memory.

330 'CANNOT SET TRACE , PROCESS NOT AT A BREAKPOINT'

The trace breakpoints, BE, BX, and BO, may only be defined/activated for processes currently at a breakpoint. Note that BA breakpoints may be defined/activated any time, regardless of the current breakpoint status of the process.

331 '"<ad>" IS NOT A PROCESS ACCESS DESCRIPTOR'

This error is in response to a "STACK OF <ad>" command and the <ad> does not refer to a process. To find out what kind of object the <ad> does point to, type either "<ad>", which will display the object and the default template name selected, or type "<ad>'SD" which will display the associated object table entry with the AD.

332 'TAB FORMAT: "nT" NOT SUPPORTED, X FORMAT USED"

This error is in response to an attempt to input "nT" as part of a display list in a template field. The format is not supported.

333 'X FORMAT NUMBER TOO BIG (255 IS MAX), 1 USED'

The maximum value permitted when using the "X" notation to indicate blanks is 255. If more is needed (very unlikely) the field may be repeated (e.g., using the repeat count) as many times as is desirable.

335 'NO PROCESSES TO RESUME'

336 'BAD MEMORY. ILLEGAL CONTEXT DATA SEGMENT ADDRESS'
337 'BAD MEMORY. ILLEGAL DOMAIN ADDRESS'
338 'BAD MEMORY. ILLEGAL CONTEXT OBJECT ADDRESS'
339 'BAD MEMORY. ILLEGAL PROCESS AD FOR BREAKPOINT'
340 'BAD MEMORY. ILLEGAL PROCESS DS FOR BREAKPOINT'
341 'BAD MEMORY. ILLEGAL CURRENT CONTEXT FOR BREAKPOINT'
342 'BAD MEMORY. ILLEGAL PREVIOUS CONTEXT FOR BREAKPOINT'
343 'BAD MEMORY. ILLEGAL CONTEXT AD FOR BREAKPOINT'
344 'BAD MEMORY. ILLEGAL CONTEXT DS FOR BREAKPOINT'
345 'BAD MEMORY. ILLEGAL DOMAIN AD FOR BREAKPOINT'

The error messages 336 - 345 imply that the memory image has been corrupted. These errors happen between the time that the debugger discovers a process at a breakpoint and the time when the breakpoint is announced on the CRT. During this time, the debugger is performing a variety of internal bookkeeping functions using information from the process and has run across inconsistent values in memory.

Unfortunately, these errors occur when the debugger is attempting to get the necessary information to announce the breakpoint to the user. Since the debugger discovers bad memory, it will not attempt to announce the breakpoint.

346 'ILLEGAL EXPRESSION FOR TOP OF MEMORY'

The INIT SYSTEM command requires the physical address of the top of memory as a parameter. The debugger will restrict all physical addresses to be below this value, until another INIT SYSTEM, INIT, or DEBUG command is given. The debugger uses the 256 bytes just below the top of memory for operating the 43203 component.

347 'FIELD "<name>" NOT FOUND'

When using the field of a reference in an expression or as the destination of the ":=" operator (i.e., to modify memory), this error will appear if the field is not part of the reference. This may occur if the field is part of a variant that is not accessible due to the current value of the disciminant:

```
?template foo is
??val: [0,8] is 0u,/;    -- displays the discriminant, [0,8]
??case [0,8] is
?? when 1 =>
??  baz: [1,8] is 0u,/;
?? when others =>
??  gorn:[2,8] is 0u,/;
??end case
??end
?!5:foo   -- display the template, show what's there
val:  5
gorn: 7   -- notice the field "baz" did not display
?p is !5:foo
?p.baz := 0   -- will get error 347
```

348 'ILLEGAL MEMORY REFERENCED BY FIELD "<name>"

This error occurs when all of the memory referenced by a field of a template is beyond the end of the actual memory present. This error happens when the template field is being used either as the destination of the ":=" operation or as part of an arithmetic expression.

Whenever a template is used in conjunction with a logical address, the debugger limits the actual memory the template can use to the memory between the logical address and the end of the segment referenced by the logical address. If template field references memory outside of that range (when it is used in an expression or as the destination of the ":=" operator), this error will be displayed.

349 'BREAKPOINT INCOMPATIBLE WITH PREVIOUSLY DEFINED ONE(S)'

The debugger uses the trace mode of 432 processes to accomplish the setting of certain breakpoints (e.g., BO inst, BE 4^5). This trace mode only supports one setting: full trace, flow trace, fault trace, or no trace. Therefore, for one process it is not possible to set breakpoints that require a combination of settings. The combinations not allowed are:

BO INST	not permitted with	BE/BX
BO FAULT	not permitted with	BE/BX

and only one "BO" breakpoint may be set for a process at any one time.

350 'TOO MANY TRACE DEFINITIONS SET FOR PROCESS'

The maximum number of breakpoints that the user may set and have active is 32. If an attempt is made to set a thirty third active breakpoint, this error may be displayed.

351 'BAD MEMORY: PROCESS GLOBALS ACCESS LIST WAS NOT FOUND'
352 'BAD MEMORY: CANNOT FIND DEBUGGER STATUS AD IN PROCESS GLOBALS'
353 'BAD MEMORY: CANNOT ACCESS DEBUGGER STATUS DS'
354 'BAD MEMORY: CANNOT CLEAR BREAKPOINT'

These errors (351 - 354) indicate that the 432 memory image has been corrupted. The Series III resident debugger, while attempting to set or clear a breakpoint, has come across an apparent inconsistency in the process/breakpoint structure.

Error 351 indicates that the process does not have a Global access segment AD. Error 352 indicates that the fifth slot in the global access segment is null. Error 353 indicates that segment indicated by the fifth AD in process globals is not a data segment AD.

356 'ONLY "BA" BREAKPOINTS MAY BE SET FOR ALL PROCESSES'

The "OF <process_list>" clause of a breakpoint may only read "OF ALL" for Break At (BA) breakpoints. For any of the tracing kinds of breakpoints, BO, BE, or BX, only breakpointed processes may appear in the "OF <process_list>" clause:

```
?examine
    PROCESSES                   CONTEXT
    9^16                        3^1F
CP: 9^1B                        5^1
    9^21                        7^28
?bo call of 9^21,9^1b    -- ok
?bo call of all          -- causes error 356
```

357 'BAD MEMORY. ILLEGAL CONTEXT AD "<ad>" ON THE STACK'

This error typically occurs when the STACK command is issued for an executing process. The reason for the error is that the call stack for the process is probabaly being updated by a processor at the same time that the debugger is attempting to display it.

This error occurs when the "previous" field of a context_AS points to a segment that is not a context access segment.

358 '"<addr>" MAY NOT BE USED AS AN INSTRUCTION ADDRESS'

Because it is not a logical address.

359 'INPUT LINE TOO LONG. INCLUDE ABORTED

This error should only happen if the include file is a binary file. The error occurs when there are too many characters between end of lines.

360 '"<file>" IS A REVISION FILE AND CANNOT BE LOADED'

The Linker can produce two outputs, a loadable EOD and a revision EOD. Error 360 is issued if the file given in a LOAD or DEBUG command was a revision EOD, not a loadable EOD.

361 'NO CURRENT PROCESS'

This error occurs in response to one of the stack commands, UP, DOWN, TOP, BOTM, or STACK, when there is no current process.

362 'OUT OF MEMORY BOUNDARIES'

This error is announced whenever an attempt is made to address memory that, according to the debugger's information, is not really present in the EV system. The "typical" cause of this error is when memory is corrupt or uninitialized. For example the DEBUG command might result in this error if memory has not been previously loaded. Another example of when this error might occur is after pulling a memory board from the EV and then attempting to load memory via any of DEBUG, LOAD, or RESTORE, but not having enough memory to hold the desired image.

The VERSION command may be used to get the debugger to print out its current assumptions about the size of memory. Either the INIT command (not INIT SYSTEM) or the DEBUG command will cause the debugger to find the Top Of Memory.

APPENDIX G
FORMAL DEFINITION OF UPDATE-432 COMMAND SYNTAX

This appendix contains a formal definition of the UPDATE-432 command syntax (including the relevant portions of the RUN command syntax). The definition uses a variant of Backus-Naur Form (BNF). The following conventions are used:

<identifier>   An identifier in angle brackets is expanded in another line. E.g. <template_definition>

upper_case_ids   Keywords are in upper case.
E.g. DEBUG, TEMPLATE, ALL

lower_case_ids   Lower case identifiers denote lexical classes
E.g. identifier, file_name

"abc"   Character strings in double quotes stand for literal items. E.g. ")", "=>"

[...]   Square brackets enclose optional items.

(...)   Parenthesis enclose several items; one of these items must be used.

{ ... }   Braces surround an item or set of items which may be repeated zero or more times.

--   A double hyphen precedes comments

a ¦ b   A vertical line denotes exclusive or.

!   Concatenate the characters on either side of the exclamation point.

The following is the definition of the UPDATE command line:

```
<UPDATE Command> ::= [<Dev>]"RUN" [<Dev>]"UPDATE" <Command Tail> cr_lf

<Command Tail> ::= [<Dev>]<Path Name> [<Control List>]

<Control List> ::= {<Control>} [<Continued Line> ¦ <NC Comment>]

<Continued Line> ::= "&" [{ ascii }] cr_lf <Control List>

<NC Comment> ::= ";" [{ ascii }]
```

```
<Control> ::= "REVISION" "(" <Path Name> ")"
              | "NEW" "(" <Path Name> ")"

<Path Name> ::= <Dev> <Filename>

<Dev> ::=   -- a valid ISIS device name of the form :fn:, where
          -- n is an integer between 0 and 9

<Filename> ::=  -- a valid ISIS filename of the form name.ext

ascii ::=  -- an ASCII character other than a cr_lf

cr_lf ::=  -- a carriage-return line-feed pair
```

APPENDIX H
UPDATE-432 ERROR MESSAGES

When UPDATE-432 detects an error, it sends a message to the user identifying the nature of the error. This message contains the following information:

A. The *class* of the error. There are three classes of errors:

1. Warnings
2. User Fatal Errors
3. Updater Internal Errors

Errors belonging to classes 2 and 3 terminate execution; class 1 errors, however, do not abort the Updater.

B. The *type* of the error. There are five types of errors:

1. EOD Errors
2. Object Manipulation Errors
3. Syntax Errors
4. UDI Errors
5. Generic Errors (i.e. all others)

C. The *exception code* (a unique number identifying the error) and the *message text* (occasionally including supplementary information unique to each error). The following exception-code/ message-text combinations may be received (excluding UDI exceptions).

Code	Message Text
F001	SOG STACK OVERFLOW
F002	MODULE NOT FOUND IN REVISION EOD
F003	OTM ENTRY DOES NOT MATCH ANY SUCH OBJECT IN PMS
F004	NO OBJECTS IN PMS REFERENCED BY SUCH OBJECT TABLE
F005	BAD CONNECTION IN =WHICH_STACK_PTR= (UPDEPH)
F006	DUPLICATED STORAGE DESCRIPTOR IN AN OBJECT TABLE
F007	CAN'T FLUSH ALLOCATED MEMORY
F008	DUPLICATED OBJECT IN PMS, ONLY FIRST PROCESSED
F009	TRYING TO INSERT OBJECT IN EMPTY MODULE RING
F00A	COUNT ITEM DOES NOT FOLLOW ADDRESS ITEM
F00B	UNMATCHED MODULE SOG
F00C	CAN'T INSERT MODULE, MEMORY WON'T FLUSH
F00D	UNMATCHED SOG
F00E	ILLEGAL INDICATOR BYTE
F00F	PREMATURE END OF FILE
F010	TRYING TO POP EMPTY SOG STACK
F011	BAD CONNECTION IN =SELECT_FILE= (UPDFIL)
F012	BAD CONNECTION IN =WHICH_FILE_PTR= (UPDFIL)
F013	BAD =SEEK_FORWARD= (UPDFIL)
F014	BAD =SHORT_SEEK_BACK= (UPDFIL)
F015	NO REFERENCE ITEM AFTER UPDATE MODULE SOG
F016	UNCONTIGUOUS MODULE GROUPS
F017	MODULE SOG EXPECTED BUT NOT FOUND
F018	COORDINATES ITEM MISSING
F019	EMPTY MODULE GROUP
F01A	UNMATCHED EOG
F01B	OT-MODULE MET BEFORE P-MODULE
F01C	MISMATCHED OBJECT COUNT
F01D	BAD PREAMBLE
F01E	ORIGIN ITEM MISSING
F01F	TIMESTAMP ITEM MISSING
F020	PRINTNAME ITEM MISSING
F021	EOD EOG ITEM MISSING
F022	UNIQUE IDS DON'T MATCH
F023	BAD COMMAND LINE SCAN
F024	FILE NAME TOO LONG OR MISSING
F025	OPEN PARENTHESIS MISSING
F026	CLOSED PARENTHESIS MISSING
F027	MISMATCHED PREAMBLES
F028	BAD COMMAND
F029	FAILED TO RENAME INTERNAL FILES
F02A	NO FILE NAME WAS EXPECTED
F02B	OTM WAS EXPECTED IN =PROCESS_OTMS_IN_REOD= (UPDREH)
F02C	EOD SOG DOES NOT SPECIFY A REVISION EOD
xxxx	UNRECOGNIZED ERROR CODE

LATE DEFINITION (con't.)

ld: A *Field* has three optional parts:

neral Form: [*Label*] [*Bit_identification*] [**IS** *Display_list*]

abel: This is the name of the field. It is displayed when the template is used to examine memory, unless the :: notation is used. A field is referenced by using its name (e.g., cc.domain).

eneral Form: $id\left\{\begin{array}{l}:\\ ::\end{array}\right\}$

it_identification: Describes which bits are to be displayed by the field.

eneral Form:

$$[@\ number\ .]\ldots\left\{\begin{array}{l}[\ Texp^{\dagger}\ [:\ Texp]\ ,\ Texp\ ]\\ @\ number\ [\mathbf{ACCESS}\ id]\\ id\end{array}\right\}$$

isplay_list: The list of conversion and editing specifications to be followed for this field. The specifications are performed from left to right.

eneral Form: *Display_elem* [, *Display_elem*]...

Display_elem: Either gives a method to translate bits into ascii characters, or describes the actual ascii characters to be printed on the CRT. A *Display_elem* may be prefixed with a repeat count.

General Form:

$$[<\ Texp^{\dagger}>]\ \left\{\begin{array}{l}Base^{\ddagger}\mathbf{U}\quad :\ Width^{\ddagger}\\ Base\mathbf{S}\quad :\ Width\\ Enumeration^{\dagger\dagger}\\ \mathbf{ASCII}\\ id\\ /\quad \text{-- newline}\\ string\\ number\mathrm{10}\mathbf{X}\\ [\ Display\_list\ ]\end{array}\right\}$$

‡*Base* and *Width* are decimal numbers (i.e., *number*10).

††Enumeration: Translates values to text

General Form: (*Enum_item* [, *Enum_item*]...)

Enum_item: [*number* =>] $\left\{\begin{array}{l}id\\ string\end{array}\right\}$

permissable expressions inside of templates.

Form:

[*Texp* { ± }]... *Tterm*

: $\left[Tterm\ \left\{\begin{array}{l}*\\ /\end{array}\right\}\right]\ldots Tprimary$

ary: $\left\{\begin{array}{l}id\\ number\\ (\ Texp\ )\\ \mathbf{BY\_P}\\ \mathbf{BI\_P}\end{array}\right\}$

intel®

DEBUG-432 REFERENCE CARD

Order Number: 172097-001

CONVENTIONS

BOLDFACE	-- keywords and punctuation to be entered verbatim (the debugger is case insensitive)
italics	-- variable information
[ ]	-- indicate an optional field
...	-- previous field may be repeated
{ }	-- one and only one field must be selected

CONTROL CHARACTERS

RUBOUT	Delete preceding character
CTRL-D	Interrupt 432 debugger execution and enter DEBUG-86
CTRL-Q	Resume console display
CTRL-R	Redisplay current line or previous command line
CTRL-S	Suspend console display
CTRL-X	Delete all characters since last carriage return
CTRL-C	Return to debugger command mode, 432 I/O disabled
CTRL-B	Enter mode which allows 432 I/O and debugger commands. (Precede each 432 input line with **%**)
CTRL-O	Enter mode which allows only 432 I/O.

THE COMMAND LINE

General Form: *Command* [; *Command*]...<CR>

<CR>	Carriage return. Terminates a command or comment.
--	Terminates a command line and starts a comment.

SYSTEM CONTROL

INIT	[**SYSTEM** *P_addr*]	-- initialize the 432 hardware
LOAD	*s-III_file*	-- load a linked 432 program
DEBUG	[*s-III_file*]	-- [load file and] enable logical -- addressing
START	[*number*]	-- start GDP *number*
IPC	$\left\{\begin{matrix} \textit{Expr1} \\ \textbf{ALL} \end{matrix}\right\}$, *Expr2*	-- broadcast *Expr2* to processor *Expr1* -- broadcast *Expr2* to all processors

ENVIRONMENT CONTROL

EXIT		-- exit the 432 debugger
INCLUDE	*s-III_file* [**LIST**]	-- include a file of debugger -- commands
BASE	[*number*]	-- display (or set) the default output -- base
SUFFIX	[*number*]	-- display (or set) the default input -- base
LOG	[*s-III_file*]	-- log all CRT interactions on a file
>**LOG**		-- direct output to the log file only
>**CRT**		-- redirect output to the CRT only
MODE		-- display the mode set by **CTRL-C,** -- **CTRL-B, CTRL-O**
VERSION		-- display debugger version number -- and status

ADDRESSING (i.e., *Addr*)

There are three kinds of addresses:

NAME	**KIND**	**GENERAL FORM**†
P_addr	Physical	**!** *number*
I_addr	Interconnect	**!!** *number*
L_addr	Logical	*number* ↑ *number* $\left\{\begin{matrix} \textbf{.}\ \textit{number} \\ \textbf{!}\ \textit{number} \end{matrix}\right.$

†Any expression which evaluates to an integer may be ι place of a *number*, however it must be enclosed in parentl

An *id* which has type **REFERENCE** may be used anywhere an *A* pears, but not where *P_addr*, *I_addr*, or *L_addr* are used.

MEMORY EXAMINATION

General Form: *Addr* [**:** *id1*] $\left\{\begin{matrix} [\textbf{.}\ \textit{id2}]... \\ \left[\begin{matrix} \textbf{LENGTH}\ \textit{number} \\ \textbf{ALL} \end{matrix}\right] \end{matrix}\right\}$

Addr: Where examination is to begin.

id1: Template name. If absent, a default will be selected.

id2: Template field name(s). If present, the rightmost field displayed. Otherwise, the entire template is used.

LENGTH *number*: How many times to re-apply template. If **LENGTH 1** is used.

ALL: Re-apply template until entire segment (or 64K) is dis

Examples:

!18:descr	-- starting at byte 18 of memory, apply templ -- descr
1↑1	-- use the default to display segment 1↑1
!!4:b16	-- display bytes 4 and 5 of interconnect spac -- (register 2)
17↑3:mem all	-- using template mem, display all of segmer
cc:rad len 8	-- show full rights of first 8 ADs of current cc
cp.status	-- display the status field of the current proc

BREAKPOINTS

General Form: [*id* **:**] $\left\{\begin{matrix} \textbf{BA}\ \textit{Addr} \\ \textbf{BE}\ \textit{Addr} \\ \textbf{BX}\ \textit{Addr} \\ \textbf{BO}\ \textit{Event} \end{matrix}\right\}$[**OF** *Process_list*]†

id: Breakpoint name. If absent, debugger will select a nar

Addr: **BA** breakpoints use an instruction address. If the tion bit offset is absent, the default is taken from struction segment header.

BE and **BX** breakpoints use an instruction segr domain AD.

Event: $\left\{\begin{matrix} \textbf{FAULT} \\ \textbf{INST} \\ \textbf{CALL} \\ \textbf{RET} \end{matrix}\right\}$

Process_list: $\left\{\begin{matrix} \textit{Addr}\ [\textbf{,}\ \textit{Addr}]... \\ \textbf{ALL} \end{matrix}\right\}$ -- process AD(s)
-- only used with **BA**

†The [**OF** *process_list*] field defaults to CP (current proce

ACTIVATE $\left\{\begin{matrix} \textit{id} \\ \textbf{ALL} \end{matrix}\right\}$	-- *id* is breakpoint name
DEACTIVATE $\left\{\begin{matrix} \textit{id} \\ \textbf{ALL} \end{matrix}\right\}$	-- *id* is breakpoint name
RESUME [*Process_list*]	-- restart the processes (defaults
EXAMINE	-- list the broken processes
SELECT *Addr*	-- select a new default process ((

ADDRESSING (i.e., *Addr*)

There are three kinds of addresses:

NAME	KIND	GENERAL FORM[†]
P__addr	Physical	**!** *number*
I__addr	Interconnect	**!!** *number*
L__addr	Logical	*number* ↑ *number* { . *number* / ! *number* }

[†] Any expression which evaluates to an integer may be used in place of a *number*, however it must be enclosed in parentheses.

An *id* which has type **REFERENCE** may be used anywhere an *Addr* appears, but not where *P__addr*, *I__addr*, or *L__addr* are used.

MEMORY EXAMINATION

```
                           ⎧[. id2]...      ⎫
General Form:  Addr [: id1]⎨⎡LENGTH number⎤ ⎬
                           ⎩⎣ALL          ⎦ ⎭
```

Addr: Where examination is to begin.

id1: Template name. If absent, a default will be selected.

id2: Template field name(s). If present, the rightmost field will be displayed. Otherwise, the entire template is used.

LENGTH *number*: How many times to re-apply template. If absent, **LENGTH 1** is used.

ALL: Re-apply template until entire segment (or 64K) is displayed.

Examples:

```
!18:descr       -- starting at byte 18 of memory, apply template
                -- descr
1↑1             -- use the default to display segment 1↑1
!!4:b16         -- display bytes 4 and 5 of interconnect space
                -- (register 2)
17↑3:mem all    -- using template mem, display all of segment 17↑3
cc:rad len 8    -- show full rights of first 8 ADs of current context
cp.status       -- display the status field of the current process
```

BREAKPOINTS

```
                   ⎧BA Addr ⎫
General Form: [id :]⎪BE Addr ⎪[OF Process__list]†
                   ⎪BX Addr ⎪
                   ⎩BO Event⎭
```

id: Breakpoint name. If absent, debugger will select a name.

Addr: **BA** breakpoints use an instruction address. If the instruction bit offset is absent, the default is taken from the instruction segment header.

BE and **BX** breakpoints use an instruction segment or domain AD.

```
        ⎧FAULT⎫
Event:  ⎪INST ⎪
        ⎪CALL ⎪
        ⎩RET  ⎭
```

```
               ⎧Addr [, Addr]...⎫  -- process AD(s)
Process__list: ⎩ALL             ⎭  -- only used with BA
```

[†] The [**OF** *process__list*] field defaults to CP (current process).

```
ACTIVATE   {id / ALL}       -- id is breakpoint name
DEACTIVATE {id / ALL}       -- id is breakpoint name
RESUME [Process__list]      -- restart the processes (defaults to CP)
EXAMINE                     -- list the broken processes
SELECT Addr                 -- select a new default process (CP)
```

CALL STACK OPERATORS (Operate on CC, the curren

```
STACK [number] [OF Addr]   -- display call stack for proc
                           -- (defaults to CP)

TOP                        -- move CC to first caller on 
BOTM                       -- move CC to last called co
UP                         -- move CC to its calling con
DOWN                       -- move CC to the next calle
                           -- context.
```

MEMORY CONTENTS FILING

```
SAVE      P__addr ⎧TO P__addr    ⎫ TO s-III__file
                  ⎩LENGTH number ⎭
RESTORE s-III__file [TO P__addr ⎧TO P__addr    ⎫]
                                ⎩LENGTH number ⎭
```

DATA STRUCTURE NAME TABLE

There are three forms of invasion:

```
1) ?Reference .        2) ?Reference        3) ?R
                          ?.                   ?. 
```

Reference: Any valid memory examination command. Af ing a *Reference*, its field names are now a (and may be used to invade further).

In forms 2) and 3) the *Reference* is one debugger commanc beginning with the dot is a subsequent command.

```
PATH  -- displays the current "invasion" path
BACK  -- backs up one element in the "invasion" path
OUT   -- clears the path
```

DIRECTORY OF USER NAMES

Listing names in the directory:

```
DIR [{id / Type}]   -- displays ALL names by default
```

Type: The debugger name types are **BREAK**, **TE** **REFERENCE**, and **INTEGER.**

Removing names from the directory:

```
REMOVE {id / ALL / Type}
```

Defining user names (see also **BREAKPOINTS** and **TEMPL** **DEFINITION):**

INTEGER definition:

id : **INTEGER** [:= *Expr*]

REFERENCE definition:

```
id1 IS Addr[{: id2 / ' SD}]...   -- id2 is a template name
```

Examples of reference definitions:

```
line__12 is 9↑0b.425      -- if 9↑0b is an instruction segme
                          -- type: BA line__12 OF ALL
p is 16↑40                -- abbreviate an AD
descrip is cp'sd          -- segment descriptor for current
io__buff is 7↑10.8:mem    -- name an address-template pair
```

Y MODIFICATION

orm: *Addr* [: *id1*] [. *id2*] ... := *Expr*

Where modification is to begin.

emplate name. If absent, "B8" will be used.

emplate field name(s). If present, only the bits indicated by ie rightmost name are modified. Otherwise, all memory touched" by template *id1* (i.e., from *Addr* to high water mark) modified.

The value to be put into memory. If an *Addr*, then memory is copied (default template is the one used on left of :=).

:

```
                -- write 0FF into memory location 20 (B8 is
                -- default)
0               -- write 0 into interconnect register 2
= 3f002f        -- write AD 3↑2 (with all rights) into 4th slot
                -- of 1↑1
ls.sp := 40     -- change Stack Pointer of current context
                -- object
:= 4↑1          -- copy 32 bits from 4↑1.0 (Left Hand
                -- Template, ord, is default for 4↑1)
```

SSION SYNTAX

[*Expr* { ± }]...[{ ± }] *Term*

$$\left[\mathit{Term}\left\{\begin{array}{l}*\\ /\\ \textbf{REM}\\ \textbf{MOD}\end{array}\right\}\right]\dots\ \mathit{Factor}$$

Primary **] ... *Primary*

$$\left\{\begin{array}{l}\mathit{id}\\ \mathit{number}\\ (\ \mathit{Expr}\ )\\ \mathit{Addr}\ [:\ \mathit{id1}^{\dagger}]\ [.\ \mathit{id2}]\ \dots\end{array}\right\}$$

d2 described under **MEMORY EXAMINATION**

ATE DEFINITION

orm: **TEMPLATE** *id* **IS** *Component_list* **END**

e name of the template

nent_list: The fields and variant parts of the template:

l Form: *Field* [; *Field*]...[*Variant_part*]

nt_part: Permits conditional display. Similar to an ADA variant record.

ral form:

SE *Bit_identification* **IS**

$$\left[\begin{array}{l}\textbf{WHEN}\ \mathit{Choice}\ [\,|\ \mathit{Choice}]\dots\ \Rightarrow\\ \quad[\mathit{Component\_list}]\end{array}\right]\dots$$

D CASE

_identification: The discriminant. A *Bit_identification* is defined in the description of *Field*

oice: The bits from the discriminant are treated as a number and compared against the *Choices*. A match selects the *Component_list* of the **WHEN.**

neral form:

$$\left.\begin{array}{l}\mathit{number}\\ \mathit{number}\ ..\ \mathit{number}\\ \textbf{OTHERS}\end{array}\right\}\begin{array}{l}\text{-- }\mathit{number}\text{ may be signed (e.g.,}\\ \text{-- }-8..-5\text{). However, negative}\\ \text{-- }\mathit{Choices}\text{ will only match 32-bit}\\ \text{-- discriminants.}\end{array}$$

TEMPLATE DEFINITION (con't.)

Field: A *Field* has three optional parts:

General Form: [*Label*] [*Bit_identification*] [**IS** *Display_list*]

Label: This is the name of the field. It is displayed when the template is used to examine memory, unless the :: notation is used. A field is referenced by using its name (e.g., cc.domain).

General Form:

$$\mathit{id}\left\{\begin{array}{l}:\\ ::\end{array}\right\}$$

Bit_identification: Describes which bits are to be displayed by the field.

General Form:

$$[@\ \mathit{number}\ .]\dots\left\{\begin{array}{l}[\ \mathit{Texp}^{\dagger}\ [:\ \mathit{Texp}]\ ,\ \mathit{Texp}\ ]\\ @\ \mathit{number}\ [\textbf{ACCESS}\ \mathit{id}]\\ \mathit{id}\end{array}\right\}$$

Display_list: The list of conversion and editing specifications to be followed for this field. The specifications are performed from left to right.

General Form: *Display_elem* [, *Display_elem*]...

Display_elem: Either gives a method to translate bits into ascii characters, or describes the actual ascii characters to be printed on the CRT. A *Display_elem* may be prefixed with a repeat count.

General Form:

$$[<\mathit{Texp}^{\dagger}>]\left\{\begin{array}{l}\mathit{Base}^{\ddagger}\textbf{U}\ \ :\ \mathit{Width}^{\ddagger}\\ \mathit{Base}\textbf{S}\ \ :\ \mathit{Width}\\ \mathit{Enumeration}^{\dagger\dagger}\\ \textbf{ASCII}\\ \mathit{id}\\ /\ \ \text{-- newline}\\ \mathit{string}\\ \mathit{number}10\textbf{X}\\ [\ \mathit{Display\_list}\ ]\end{array}\right\}$$

‡*Base* and *Width* are decimal numbers (i.e., *number*10).

††Enumeration: Translates values to text

General Form: (*Enum_item* [, *Enum_item*]...)

$$\mathit{Enum\_item}:\quad[\mathit{number}\ \Rightarrow]\left\{\begin{array}{l}\mathit{id}\\ \mathit{string}\end{array}\right\}$$

†*Texp*: permissable expressions inside of templates.

General Form:

Texp: [*Texp* { ± }]...*Tterm*

$$\mathit{Tterm}:\quad\left[\mathit{Tterm}\left\{\begin{array}{l}*\\ /\end{array}\right\}\right]\dots\mathit{Tprimary}$$

$$\mathit{Tprimary}:\quad\left\{\begin{array}{l}\mathit{id}\\ \mathit{number}\\ (\ \mathit{Texp}\ )\\ \textbf{BY\_P}\\ \textbf{BI\_P}\end{array}\right\}$$

intel®

Cross Development Syste
Workstation User's Guid

172097-00

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Int product users. This form lets you participate directly in the publication process. Your comments will hel us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness (this publication. If you have any comments on the product that this publication describes, please conta your Intel representative. If you wish to order publications, contact the Intel Literature Department (se page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions f improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ____________ DATE ____________

TITLE ____________

COMPANY NAME/DEPARTMENT ____________

ADDRESS ____________

CITY ____________ STATE ____________ ZIP CODE ____________
(COUNTRY)

Please check here if you require a written reply. □

Ɔ LIKE YOUR COMMENTS . . .

ɔocument is one of a series describing Intel products. Your comments on the back of this form ıelp us produce better manuals. Each reply will be carefully reviewed by the responsible ın. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
SSO Technical Publications, WW1-487
3585 SW 198th Ave.
Aloha, OR 97007

INTEL CORPORATION, 3585 S.W. 198th Avenue, Aloha, Oregon 97007 • (503) 681-8080

Printed in U.S.A./Y100/5K/01/04/82/AP