symbolics™

3600 Technical Summary

February 1983
Cambridge, Massachusetts

This document corresponds to the Symbolics 3600 system as of February 1983.

This document was prepared by the Documentation and Education Services Department of Symbolics, Inc.

Principal writer: Curtis B. Roads

The information in this document is subject to change without notice and should not be construed as a commitment by Symbolics, Inc. Symbolics, Inc. assumes no responsibility for any errors that may appear in this document.

Symbolics, Inc. makes no representation that the interconnection of its products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of a license to make, use, or sell equipment constructed in accordance with its description.

The terms and conditions governing the sale of Symbolics hardware products and the licensing of Symbolics software consist solely of those set forth in the written contracts between Symbolics and its customers. No representation or other affirmation of fact contained in this document, including but not limited to statements regarding capacity, response-time performance, suitability for use or performance of products described herein, shall be deemed to be a warranty by Symbolics for any purpose, or give rise to any liability of Symbolics whatsoever.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc.

Symbolics, Inc. assumes no responsibility for the use or reliability of its software on equipment that is not supplied or maintained by Symbolics, Inc.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.
MACSYMA is a trademark of Symbolics, Inc., Cambridge, Massachusetts.
DEC, Digital, TOPS-20, VAX, and VMS are trademarks of Digital Equipment Corporation.
TENEX is a registered trademark of Bolt Beranek and Newman Inc.
UNIX is a trademark of Bell Laboratories, Inc.

Copyright © 1983, Symbolics, Inc. of Cambridge, Massachusetts.
All rights reserved. Printed in USA.
This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Table of Contents

Preface 1

3600 Overview 3

History of the Lisp Machine Concept 4
3600 Software Overview 5
Why Lisp? 5
Zetalisp 5
The Lisp Environment 6
Network Software 7
3600 Hardware Overview 7
The 3600 System 7
The 3600 Processor 8
3600 Networks 8
3600 Peripherals 10
The User Interface 10
Notation Conventions 12

3600 Software: Development Tools 13

The Zmacs Editor 13
Manipulating Text 14
Editing Lisp Programs 15
Interacting with the Lisp environment 15
Using the Mouse 16
Bit-mapped Display Editing 17
Zmacs Commands 17
Customizing the Editing Environment 18
Other Program Development Tools 19
The Inspector 19
The Display Debugger 21
Peek: Examining the State of the System 23
The File System Editor: FSEdit 25
The Font Editor: FED 25
Tools for Managing Large Systems 28
Defining a System: Selective Modification and Recompilation 28
Patching Bugs 29

3600 Software: Languages 31

The Zetalisp Language 31
Data Types 31
Program Control Mechanisms 32
Function Calling 34
Multiple Values 34
Input/Output Mechanisms 34
Predefined Functions 36

Packages for Independent Namespaces 36
Interactive Debugging Tools 37
Flavor System: Language features to support object-oriented programming style 38
Signalling and Handling Conditions 44
Macros: Extending the Language 45
Access to System Subprimitives 46
Common Lisp Compatibility **46**
Other Supported Languages and Systems **47**
Interlisp Compatibility Package 47
FORTRAN 77 Tool Kit 49
LIL 50
MACSYMA 52

3600 Software: Environment **55**

The Window System: The User's Perspective **55**
A Hierarchical Window System 56
Menus and Choices 56
The Mouse Documentation Line 57
The Status Line 58
Multiple Display Screens 58
The Window System: The Programmer's Perspective **59**
Primitive Graphics Operations 59
Choice Facilities 59
Scrolling 61
Creating New Windows with Mixin Flavors 62
File Systems **62**
The Lisp Machine File System 62
File System Reliability 63
Remote File Systems 64
Virtual Memory: The Programmer's Perspective **65**
Scheduling Processes **66**

3600 Software: Network Communications **69**

Electronic Mail: Zmail and Converse **69**
Reading and Answering Mail 69
Selecting and Filtering Mail 71
Customizing Zmail 71
The Converse Utility for Interactive Messages 73
Network Software **73**
Ethernet Support 73
Remote login 75
Symbolics Auto-dial Feature 75

3600 Hardware: A New Lisp Machine **77**

3600 Hardware: Processor Architecture **79**

Tagged Architecture **79**
Run-time Data-Type Checking 79
Word Formats and the Cdr-coding Feature **80**
Cdr-coding for List Compaction 81
Hardware-supported Data Types **82**
Stack Mechanisms **83**
Hardware Support for Stack Groups 84
3600 Instruction Set **85**
Instruction Formats 85
The Instruction-Execution Engine **88**
Example of Instruction Execution: **ADD** 88
The Instruction Fetch Unit **89**
Microcode and Microtasks **89**
Microtasking 89

3600 Hardware: Organization of Memory 93

Instruction Cache 93
Stack Buffers 93
Hardware Pointers 93
Physical Memory 94
Virtual Memory 94
Virtual Memory Operation 94
Translating a Virtual Address into a Physical Address 96
Garbage-Collection Mechanisms 98
Hardware-assisted Garbage Collection 98
The L Bus 100
Block-Mode Operation 101
Direct Memory Access (DMA) Operation 101
The L Bus Clock 103

3600 Hardware: I/O Systems 105

The FEP and MULTIBUS 105
Serial Lines 105
MULTIBUS Interrupts 106
Bootstrap Loading the 3600 106
Hardware Error Handling 106
Running Diagnostics from the FEP 107
The Spy bus 107
The FEP File System 108
The NanoFEP 108
The 3600 Console 108
Digital Audio Output System 109
Disk Controller 110

3600 Hardware: Packaging and Specifications 113

Processor and Console Cabinets 113
Electrical Specifications 114
Environmental Specifications and Requirements 115
Physical Dimensions and Weights 115

3600 Hardware: Peripherals 117

Disk Systems 117
Ethernet Interface 117
Color Display System 118
Color Memory Addressing Modes 120
Color Display System Options 121
Tape Drives 124
Laser Graphics Printer 124

3600 Technical Communication 127

Glossary 131

Index 139

List of Figures and Tables

Figure 1.	Block diagram of a 3600 network.	9
Figure 2.	The 3600 console display, keyboard, and mouse.	11
Figure 3.	An Inspector window.	20
Figure 4.	A Display Debugger window.	22
Figure 5.	A Peek window, showing the state of current windows.	24
Figure 6.	A File System Editor window, with a momentary menu of directory operations.	26
Figure 7.	The font editor screen.	27
Figure 8.	A package hierarchy.	37
Figure 9.	Flavor combination (1) and instantiation (2).	40
Figure 10.	Message-passing operation. Flavor instance **x** sends a message to flavor instance **y.**	41
Figure 11.	The inheritance network of the flavor **tv:menu** built by combining many component flavors.	43
Figure 12.	A MACSYMA window in use, example of the **PLOT3D** function.	53
Figure 13.	Example of a momentary menu.	57
Figure 14.	Example of a choose-variable-values menu.	57
Figure 15.	The mouse documentation line and the status line.	58
Figure 16.	The double-arrow symbol indicates that scrolling mode is in effect. The thickened line is a proportional representation showing the current screen's size and position in the entire buffer.	61
Figure 17.	Kinds of file systems.	62
Figure 18.	Zmail input and output diagram.	70
Figure 19.	Zmail window in a mail-reading state.	72
Figure 20.	3600 word formats.	81
Figure 21.	Representation of the list (A B C) in normal form uses six words. (Note that the type field is not shown in this diagram.)	82
Figure 22.	Representation of the list (A B C) in compacted (cdr-coded) form uses three words. (Note that the type field is not shown in this diagram.)	83
Figure 23.	Instruction pipeline path, showing the relation between the IFU and the rest of the processor.	90
Figure 24.	3600 virtual memory mechanisms.	95
Figure 25.	View of the L bus backplane.	100
Figure 26.	L bus timing.	102
Figure 27.	Basic digital audio output system.	111
Figure 28.	Front view of the board layout on the backplane. An additional cabinet is provided for memory expansion beyond that shown here.	114
Figure 29.	The 169-Mbyte disk drive.	118
Figure 30.	The 474-Mbyte disk drive.	119
Figure 31.	Color memory topology in the Plane addressing mode.	122
Figure 32.	Color memory topology in the Pixel addressing mode.	122
Figure 33.	Color memory topology in the Packed addressing mode.	123
Figure 34.	Color memory topology in the Fill addressing mode.	123
Figure 35.	The TD20 cartridge tape drive.	124
Figure 36.	The TD80 streaming tape drive.	125
Table 1.	Instruction Categories. (Note: The instructions listed here do not constitute the entire instruction set.)	86

Preface

This technical summary is a detailed introduction to all aspects of the Symbolics 3600[1] system. It is addressed to technical managers, programmers, and computer system specialists who are already familiar with computer software, hardware, and terminology.

The text begins by outlining the history of the Lisp Machine concept, its development at the Massachusetts Institute of Technology (M.I.T.), and its evolution at Symbolics, Inc. Next are overviews of the 3600 software and hardware. For a general description of the 3600 system, read only these overviews.

After the overviews, this document is divided into four software chapters:
- 3600 Software: Development Tools
- 3600 Software: Languages
- 3600 Software: Environment
- 3600 Software: Communications

These are followed by five hardware chapters:
- 3600 Hardware: Processor Architecture
- 3600 Hardware: Organization of Memory
- 3600 Hardware: I/O Systems
- 3600 Hardware: Packaging and Specifications
- 3600 Hardware: Peripherals

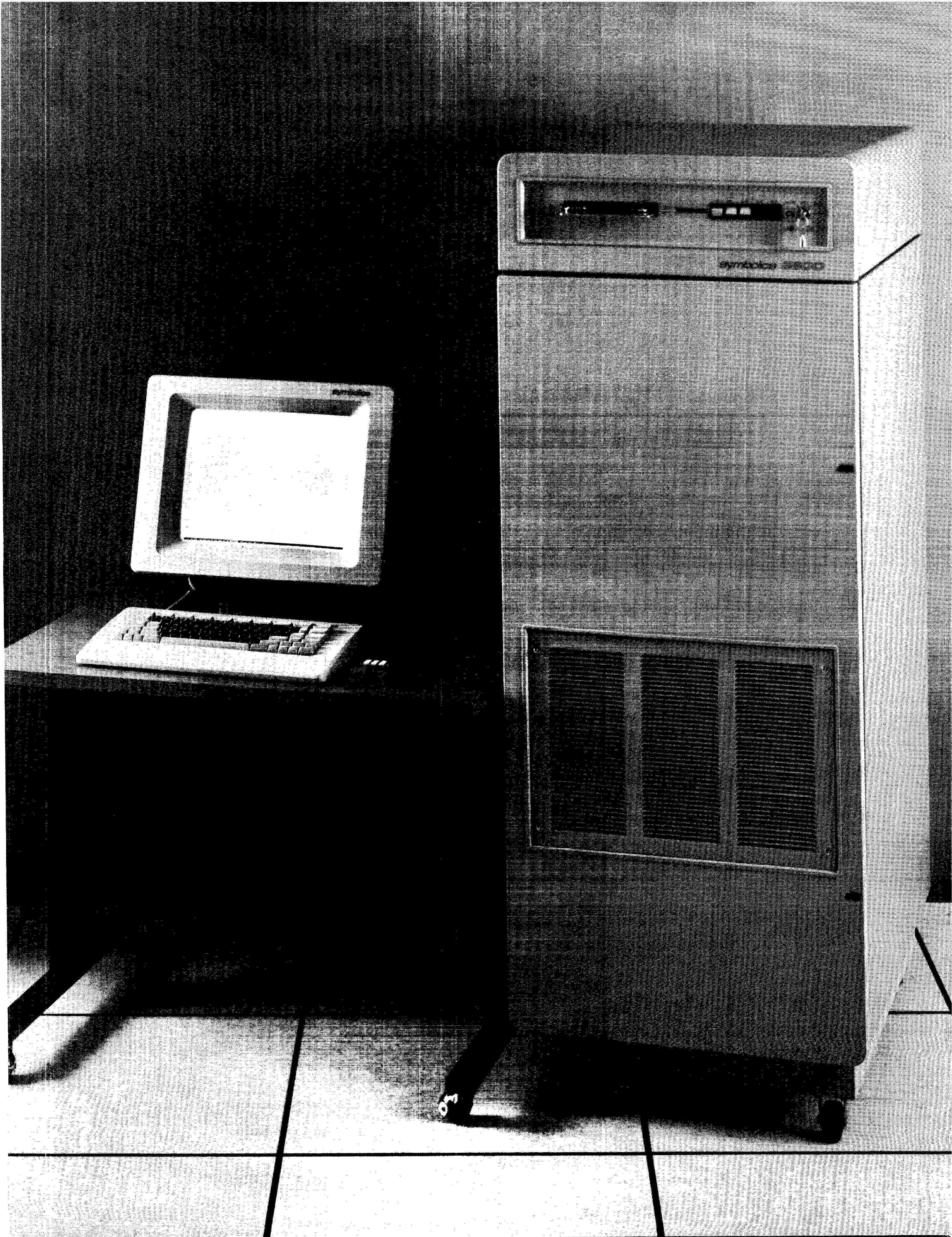
A chapter on Technical Communication completes the presentation.

Each chapter contains detailed summaries of the features of the 3600, along with technical details of how the features work.

If you are uncertain about the meaning of a particular term, consult the Glossary provided at the end of this volume. Use the Index to find the meaning of a particular term as we use it in the context of the 3600 system.

---

[1]Patent applied for.



3600 Overview

This chapter outlines the history of the Lisp Machine concept and discusses the main features of the 3600 software and hardware.

The Symbolics 3600 is a 36-bit single-user computer designed for high-productivity software development and for the execution of large symbolic programs. The 3600 processor gives the user all the computational power associated with supermini computers in a dedicated workstation. The tagged architecture of the processor allows run-time data-type checking in hardware, eliminating data declarations in programs.

The system software constitutes a large-scale programming environment, with over a half-million lines of system code accessible to the user. Object-oriented programming techniques are used throughout the 3600 system to provide a reliable and extensible integrated environment without the usual division between an operating system and programming languages.

Printed documentation is provided for users at all levels of experience. Conceptual documents present an overview of a topic, procedural guides show users how to accomplish specific tasks, and reference manuals describe the features of the system in detail. In addition, online documentation is available throughout the 3600 software environment.

Typical applications of the 3600 include research and development in the following areas:

- Artificial Intelligence (AI)
- Computer-aided Design (CAD)
- Expert Systems
- Simulation
- Signal Processing
- Education
- Physics
- Graphical Animation
- Communications
- Very-Large-Scale-Integration (VLSI) Circuit Design
- Speech Recognition and Understanding
- Pattern Recognition and Image Understanding
- Natural Language Understanding

History of the Lisp Machine Concept

In recent years, the economics of computer hardware and the computational demands made by modern software have converged to make personal, networked computers more attractive than timeshared systems. In response to this changing situation, researchers at the M.I.T. Artificial Intelligence Laboratory initiated the Lisp Machine project in 1974. The project was aimed at developing a state-of-the-art personal computer that would support programmers developing large and complex software systems. An important decision was made early in the design process: for consistency throughout the software environment, all of the system code would be written in a single language — Lisp.

The Lisp Machine concept rests on the following tenets:

- Dedicated personal computer and console
- Fast Lisp execution
- Tagged architecture (run-time data-type checking and generic instructions)
- Virtual memory
- Integrated local area network
- Interactive, high-resolution, bit-mapped graphics

As the first stage of the project, a simulator for the Lisp Machine was written on a timeshared computer system. This enabled software development to proceed while the hardware was being debugged. Software development for Lisp Machines has been ongoing since 1975. The first-generation Lisp Machine, the CONS, was running in 1976. A second-generation Lisp Machine, called the CADR, incorporated some hardware improvements. It was introduced in 1978, replacing the CONS.

In 1980, Symbolics, Inc., was formed with the purpose of combining past experience with the latest technology to develop a new line of Lisp-based computer systems and related products. Symbolics introduced a third-generation Lisp Machine, the LM-2, in 1981. The LM-2 is basically an M.I.T. CADR, repackaged for higher reliability and easier servicing. From 1979 to 1982, research continued on a much more powerful and cost-effective system. This fourth-generation Lisp Machine, known as the

3600, is based on a completely new hardware design, yet it retains software compatibility with the LM-2.

3600 Software Overview

This section describes the advantages of Lisp, specific features of Zetalisp (the dialect of Lisp currently used on the 3600), the Lisp Machine programming environment, and the network software.

Why Lisp?

Lisp was designed for symbol processing. Symbolic processing includes computation with symbols and relationships as well as numbers, characters, and bits. Most major artificial intelligence systems are written in Lisp, including programs for expert problem-solving, common-sense reasoning, learning, natural language processing, education, speech, intelligent signal processing, and vision. Lisp has many features which make it useful for symbol processing.

- Lisp is interactive.
- Lisp functions and data have the same form; programs can generate other programs and then pass control to them.
- The Lisp environment provides powerful editing and debugging tools.
- Lisp is easy to learn; the parenthesis notation makes Lisp syntax uniform.
- Lisp is extensible.

In recent years, highly efficient compilers and Lisp-oriented processors have been developed which have dispelled the earlier notion of Lisp as a slow language. Optimized code generators for Lisp have enabled its use as a systems programming and implementation language. For example, all of the system code in the 3600 is written directly in Lisp. Applications in symbolic mathematics (MACSYMA), document processing, and computer-aided design have also been developed exclusively in Lisp.

Zetalisp

Zetalisp is a Lisp dialect developed specifically for the Lisp Machine. Zetalisp is closely related to the Maclisp dialect developed in the 1970s. It is largely compatible with Maclisp, while introducing many new features and improvements. These include:

- A full range of data types, including many numerical types, lists, strings, arrays, planes, and user-defined structures.
- Modern control constructs, including a very general *loop* iteration facility, asynchronous nonlocal exits, coroutines, and processes.
- Flexible function calling and multiple-value returns.
- Stream-oriented input and output.
- The Flavor System for object-oriented programming with message-passing.
- Macros for extending the Zetalisp syntax.
- Predefined functions which support such operations as sorts, hash tables, linear equations, and matrix operations.
- Multiple name spaces (packages).

Symbolics is committed to compatibility with the soon-to-be-released Common Lisp specification. Currently under development, Symbolics Common Lisp is a compatible superset of the Common Lisp standard.

Note: In the rest of this document, "Zetalisp" and "Lisp" are used synonymously.

The Lisp Environment

The 3600 provides an extensive interactive programming environment all written in Lisp. Over a half-million lines of code are provided with the basic system (in both source and compiled form). This includes over 10,000 compiled functions to which users have full access. The software incorporates the following components:

- Flavor-based window system
- Flavor-based choice facilities, including many types of menus
- A real-time text editor with many advanced features, including interpretation and compilation of Lisp forms
- Incremental compilers
- Dynamic loading and linking
- A flexible display-oriented debugging system
- Flavor-based condition system for error-handling
- Ethernet local area network communications
- Interlisp Compatibility Package

Software options include the FORTRAN 77 Tool Kit and the MACSYMA symbolic mathematics system.

Network Software

Symbolics Network System software and hardware enable 3600s to share resources and exchange data with each other, with Symbolics LM-2s, and with standard timeshared computers running a variety of operating systems such as UNIX and VAX/VMS. Local network communications are supported via an industry-standard 10-Mbit/sec Ethernet interface.

Network connections are an important aspect of the 3600's software design. System software provides support for network operations including the following features, implemented compatibly across all supported operating systems:

- Sophisticated electronic mail utility
- Real-time interactive messages
- Generic file-system access
- Remote login capability

In addition to Ethernet software, the Symbolics Auto-dial Feature permits direct connection between Symbolics computers over standard telephone lines. It also can act as a gateway between local networks at different sites. Many high-level user services previously available only on computers linked by local area networks are provided between computers on different local networks.

3600 Hardware Overview

This section outlines the features of the 3600 processor architecture, network design, and peripherals.

The 3600 System

The 3600 hardware was designed to bring the power of mainframe computing to the individual programmer. This is made possible by:

- Dedicated 36-bit processor (32 bits data, 4 bits tag) with instruction prefetch and run-time data-type checking
- Demand-paged virtual memory (28 bits virtual, 24 bits physical), word-addressed
- Interactive high-resolution (1150 x 900 pixels) display
- Console processor based on dedicated MC68000 for handling keyboard and mouse input/output
- Optional high-resolution color display system and frame buffers (up to 1280 x 1024 x 32 pixels)

- Innovative keyboard design
- Mouse pointing device
- Built-in digital audio output subsystem
- Industry-standard MULTIBUS (IEEE 796) for peripheral expansion
- MC68000-based front-end processor (FEP) on the MULTIBUS
- Dedicated 169-Mbyte Winchester disk (standard), or 474-Mbyte Winchester disks (optional), or 300-Mbyte removable disk (optional)
- 10-Mbit/sec Ethernet transceiver and interface
- Three serial input/output lines (both high- and low-speed)

The 3600 Processor

The workhorse of the 3600 system is the central processing unit. The 3600 processor was designed along with the software, yielding an unusually close coupling between the processor and the Lisp system software. Features of the 3600 processor include:

- 36-bit internal data paths
- Tagged architecture: run-time data-type checking with no overhead
- Stack-oriented architecture, with large stack buffers (the top page of the stack is always kept in a special high-speed memory)
- Instruction prefetch unit and 2K instruction cache
- Microtasking operation (many Ethernet and disk controller functions are performed in multiplexed processor microcycles)
- Hardware-assisted garbage collection for memory efficiency
- IEEE-standard floating-point operations

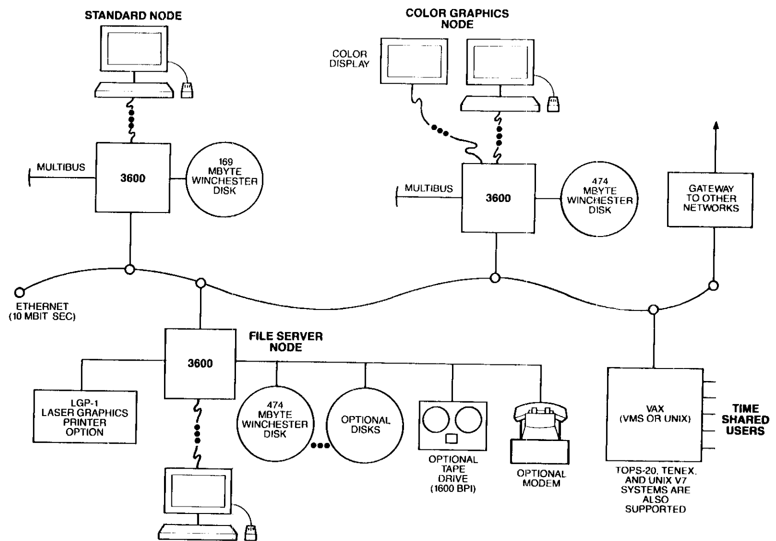
3600 Networks

Although standalone configurations are available, Symbolics 3600 computers are designed to be linked in a local area network such as Ethernet (see figure 1). Each 3600 is fitted with a 10 Mbit/sec Ethernet transceiver and cable as standard equipment.

Local area networks facilitate the sharing of programs, data files, and network resources. Users realize the best of two worlds:

- The benefits of timesharing

Figure 1. Block diagram of a 3600 network.



VAX is a trademark of Digital Equipment Corp
UNIX is a trademark of Bell Laboratories, Inc
TENEX is a trademark of Bolt Beranek and Newman Inc.

Copyright © 1983 Symbolics, Inc.

Intercommunication among users, shared access to files, use of high-quality input and output devices.

➚ The advantages of single-user stations
Fast response, dedicated processor and memory resources, customizable environment, protection from crashes, and greater availability.

As few as two or as many as one hundred 3600s can be coordinated on a single Ethernet network.

In addition, 3600s can be connected to other computer systems on the Ethernet, allowing access to a variety of file systems. Connected networks can be set up by attaching *gateways* to an Ethernet, which attach it to different Ethernets or other large-scale computer networks.

On each local network, a 3600 with a full-duplex modem with auto-dial and auto-answer capability constitutes the Symbolics Auto-dial Feature. This feature allows high-level communication between 3600s that are not on the same local area network. Using Auto-dial, customers can receive remote diagnostics and software patches from Symbolics inexpensively and expeditiously.

3600 Peripherals

3600 systems can be configured with a number of peripherals, including:
- High-resolution color display system
- Industry-standard tape drive
- Additional Winchester or removable disks
- Symbolics LGP-1 Laser Graphics Printer

The User Interface

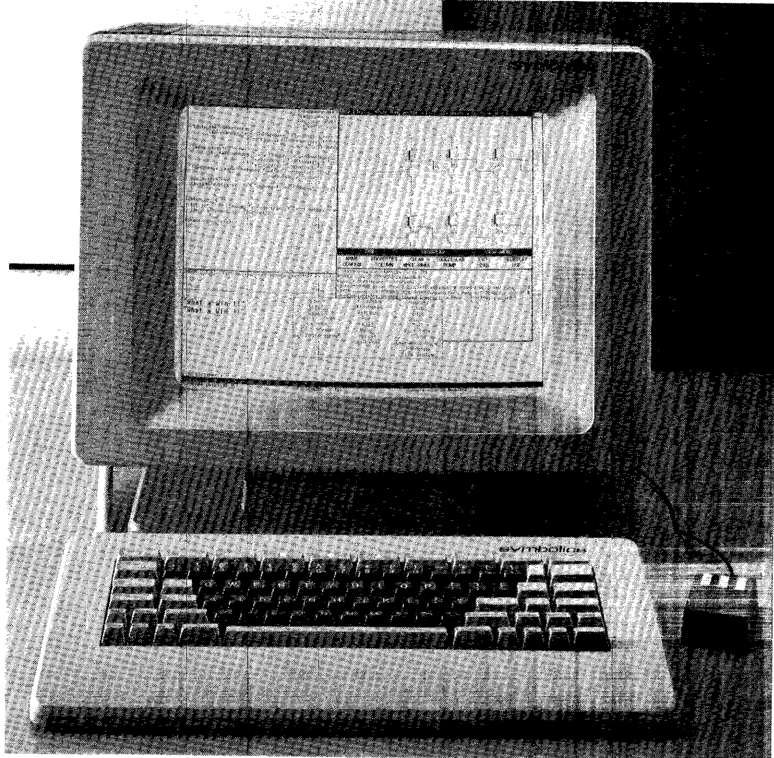
Users interact with a computer system through its *user interface* — a combination of software and hardware that determines how information is presented to the user.

The user interface of the 3600 is based on software that interacts with four hardware components:
- A large alphanumeric keyboard with 88 keys, including many special function keys.
- A high-resolution bit-mapped graphics display screen.
- The mouse, a hand-held graphical pointing device that rolls on the desk surface next to the alphanumeric keyboard. Moving the mouse on the desk causes an arrow (cursor) to move on the display screen. The mouse has three buttons which can be *clicked* (pressed) to invoke choices in various contexts.
- The digital audio output system.

The goals of the 3600 user interface software design include:
- Clarity of presentation and interaction
- Consistency of interaction protocols throughout the system
- Ready access to help and information facilities

The following features create a coherent user interface:
- Window and menu-oriented graphics
- "Modeless" context-switching (no need to kill one program to use another)

Figure 2. The 3600 console display, keyboard, and mouse.



- Consistent entering and exiting of programs
- Dynamic mouse documentation (changes depending on context)
- Dynamic help and information facilities

Notation Conventions

The rest of this document uses particular type fonts, formats, and language conventions to present technical material.

- **send**
  Printed representation of Lisp objects in running text.
- `(send p3 ':priority)`
  Lisp code examples, set off from the text.
- Find Unbalanced Parentheses, Do It
  Command names in Zmacs, Zmail, and menus appear with initial letter of each word capitalized.
- Clicking
  Pushing down one of the buttons on the mouse momentarily.

3600 Software: Development Tools

A distinguishing feature of the 3600 is the wealth of software which has been written for it. This software realizes two main functions:

- An integrated and extensible environment for program development.
- An extensive support environment for the execution of large software systems.

The Lisp program development environment consists of the Zmacs editor, Lisp, the Display Debugger, and utilities for examining Lisp objects. The editor is used to create Lisp source code and to compile functions and files. The code is used in the Lisp Listener. Errors invoke the Debugger, which is used to examine the environment. Lisp data structures are examined in the Inspector. The state of the system can be examined with Peek.

In this chapter you will find occasional references to Lisp language elements. See the next chapter, 3600 Software: Languages, in particular the section The Zetalisp Language, page 31, for definitions and more detailed descriptions of these items.

The Zmacs Editor

Zmacs is the text editor on the 3600 system. It is a real-time display editor; this means that the text being edited is always visible and commands are executed immediately. Zmacs is used for editing both text and program code. It allows users to switch between several files being edited. Special features for editing programs and for communicating with the Lisp environment are provided.

Zmacs has several levels of usage, ranging in complexity from the very simple to the advanced. A beginner can get by with a small subset of basic commands. At the same time, sophisticated commands are available for the more experienced user. An important feature of Zmacs is that users can customize its behavior and add extensions.

Zmacs exploits the mouse and the graphics capabilities of the console display. Some commands can be given by either the keyboard or the mouse. The mouse can be used to point at a particular character or graphically mark a region.

The command set of Zmacs is based on the well-known EMACS editor. Editors featuring the basic EMACS command set are now available on many timesharing systems. Zmacs uses this command set, and most EMACS commands are implemented compatibly in Zmacs. In a computer facility that includes timesharing systems with EMACS-like editors, users are able to move between the 3600 system and the timesharing systems without having to learn two different editors.

Manipulating Text

With Zmacs, inserting text is simple — just type the text. Entering a special "insert mode" is not necessary. Single-character commands move the cursor around, forward or backward or to the next or previous line. The mouse can also be used for this purpose.

With about fifteen basic commands, you can edit text effectively. With proficiency you can increase your active repertoire. Zmacs has commands to operate on the following units:

- Buffers
  Get, select, display, kill, move within
- Characters
  Move, delete, transpose
- Words
  Move, kill, transpose, change printed case
- Lines
  Move to beginning or end, open, close, kill, transpose
- Sentences
  Move to beginning or end, kill
- Paragraphs
  Move to beginning or end, mark, fill
- Lisp forms
  Mark, move, kill, fill or grind, move to beginning or end
- Screens
  Show next or previous screen, move to top, realign, redisplay
- Regions
  Mark, kill, save
- Windows
  Move to, split in two, merge into one

Operating on these units, Zmacs commands perform the following actions:

- Mark a unit of text and move it or copy it somewhere else in the file
- Search for a given unit
- Replace all occurrences of one string with another string, globally or incrementally
- Indent whole regions of text by arbitrary amounts
- Fill or justify paragraphs or arbitrary regions

For speed, frequently used commands are invoked with only one or two keystrokes. Commands that are less frequently used have longer, mnemonic names; sophisticated command completion allows typing of abbreviations for these longer names. Through the online command documentation, help is available immediately for any command in Zmacs.

The editing environment can be tailored to suit the specific needs of regular text, Lisp code, FORTRAN 77, and LIL code.

Editing Lisp Programs

Zmacs understands the syntax of Zetalisp. It signals matching parentheses by blinking on the display screen: whenever the cursor is just to the right of a close-parenthesis, the matching open-parenthesis blinks, providing instant confirmation of balanced parentheses. Zmacs ignores parentheses that are inside character strings or otherwise quoted. A command called Find Unbalanced Parentheses looks over an entire file and positions the cursor at sites of suspected parenthesis errors.

Zmacs knows a set of stylistic rules for indentation of Lisp programs. It can perform a carriage return and automatically supply the appropriate amount of indentation for new lines in a program.

Interacting with the Lisp environment

Zmacs is closely integrated with the Lisp environment; any Lisp object in the entire system can be accessed quickly. Zmacs supports the following operations on Lisp code:

- Individual function evaluation or compilation from within the editor

- Evaluation or compilation of an entire file
- Disassembly of compiled code
- Expansion of macros

Zmacs knows about many Lisp objects. This enables it to:
- Tell users the argument list of Lisp functions
- Generate descriptions of variables
- Generate descriptions of flavors
- Provide online documentation of functions and flavors
- Find modified functions

One of the most powerful commands in Zmacs, m-. (pronounced "meta-dot"), allows you to edit the definition of any Lisp function, variable, flavor, data structure, or other named object. After typing the name of the thing to be edited, or pointing with the mouse cursor at any name visible in the editor window, Zmacs finds the source text that defines that object. It automatically reads in the file containing the text, if necessary, and positions the cursor at that definition.

Zmacs can keep track of changes made to programs. The Compile Changed Definitions command will compile any code in Zmacs buffers that has been changed in an editing session.

Using the Mouse

Throughout Zmacs, the mouse provides flexibility. The mouse can be used for many editing operations. You can point with the mouse to position the cursor at a precise point in the text. You can *mark* (delineate) whole regions of the screen to be operated on by sweeping over them with the mouse.

Without typing, you can copy or move regions of text from one place to another. The mouse can be used for *scrolling*, that is, to control which portion of the file is visible on the screen. With the mouse, you can move to an arbitrary place in the file by graphically specifying how far into the file you want to see. At any time, a menu of editor commands is available.

Bit-mapped Display Editing

Zmacs takes advantage of the bit-mapped display to provide faster and more convenient interaction with the user than is possible on a conventional terminal. Zmacs can edit text written in multiple fonts, with either fixed or proportional spacing or any mixture of the two. The blinking of matching parentheses and the use of the mouse have already been mentioned. Many commands operate on a previously selected region. In Zmacs, this region is underlined on the display, eliminating the danger of operating on the wrong region.

Zmacs Commands

Zmacs has over four hundred commands. Online documentation describing the commands is available at all times.

Some of the more advanced Zmacs features include the following:

- List Definitions
  Displays the names of Lisp functions, macros, variables, structures, flavors, and methods defined in the current file. Clicking on one of these with the mouse positions the cursor to the definition of that function.
- List Combined Methods
  Understands the Flavors construct of Zetalisp. Give this command the name of a flavor and the name of a message. It types out a list of all of the component methods that are combined to form the handler for the specified message. Clicking with the mouse on any of these names finds the definition of that component method and positions the cursor there.
- Keyboard Macros
  Captures a sequence of Zmacs commands and gives that sequence a name. This new name can be invoked just like any other Zmacs command or can be assigned to a key.
- Electric Shift Lock Mode
  Facilitates entering programs that are in upper case: whenever a Lisp symbol is typed, such as the name of a function, variable, or special form, it is inserted in upper case; other characters are inserted normally.
- Sort

Sorts the text in the buffer or in a region into alphabetical order, either line-by-line or with user-specified division into records.

- Word Abbreviation
  Lets users define short abbreviations for commonly typed words or phrases. When they type in one of these abbreviations, it automatically expands into its full definition. A main application for this is as a spelling corrector.
- Auto-Fill Mode
  Automatically inserts carriage return characters as text is typed. This allows continuous typing where you need not think about inserting line breaks.

Customizing the Editing Environment

The Zmacs editing environment can be customized. Some commands have options which can be set to suit each user's style of working. You can also control the assignment of commands to keys, adding new commands and replacing other commands as you see fit. The frequent commands can be assigned to a single keystroke. New editor commands can also be written in Lisp and added to Zmacs.

Zmacs is built on a large and powerful system of text-manipulation functions and data structures, called Zwei. By writing programs that call Zwei functions to perform primitive text manipulation operations, you can build a library of commands and documentation.

Any interactive program running on the 3600 can provide the Zmacs text-editing capability simply by calling functions in the Zwei system. For example, the Zmail mail reading system uses Zwei functions to allow editing of a mail message as it is being composed, or after it has been received. This sharing of large subsystems is unique to computer systems that provide a large, dynamically linked environment, as the Lisp environment does.

Other Program Development Tools

Programmers need effective support tools in order to develop large systems and complex graphics applications. The Inspector, Display Debugger, and Peek utilities are all advanced program debugging and inspection tools that make heavy use of the 3600's graphics capabilities. The File System Editor is a tool for managing source code and text files throughout the system. The Font Editor allows users to design and modify type fonts that appear on the screen.

The Inspector

The Inspector is a graphic tool for examining data structures. It displays a Zetalisp object, showing all of its components.

- ↗ List
  Displays elements of list
- ↗ Array
  Displays elements of array
- ↗ Structure
  Displays structure slots and names of slots
- ↗ Symbol
  Displays name, value, property list, associated function, package
- ↗ Function
  Displays compiled assembly language code
- ↗ Flavor
  Displays instance size, bindings, message-handler, name, instance variables, method table, dependencies, inclusions, package, property list
- ↗ Flavor instance
  Displays flavor, information about methods, instance variable names and values

The Inspector appears as a full screen with several subpanes (see figure 3). The top pane is an interaction window. Below that is a history window and some inspection windows. Each inspection window can contain a different object. When a new object is inspected in the bottom window, the previously inspected objects get shifted to the higher windows. The history window records the objects already examined, allowing you to back up and continue down another path.

Figure 3. An Inspector window.

```
'tv:menu

                              More above                                 Exit
  #<FLAVOR TV:MENU 526661>                                               Return
  TV:MULTIPLE-MENU-CHOOSE-MENU                                           Modify
  #<FLAVOR TV:MULTIPLE-MENU-CHOOSE-MENU 555364>                          DeCache
  (TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN TV:MENU)                           Clear
  TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN                                     Set \
  #<FLAVOR TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN 555340>
                            Bottom of History
                              Top of object
a list
(TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN TV:MENU)

                            Bottom of object
                              Top of object
TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN
Value is unbound
Function is unbound
Property list: (SI:FLAVOR #<FLAVOR TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN 555340> SOURCE-FILE-NAME
Package: #<Package TV 3615215>

                            Bottom of object
                              Top of object
#<FLAVOR TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN 555340>
Named structure of type SI:FLAVOR

SI:FLAVOR-INSTANCE-SIZE:                    NIL
SI:FLAVOR-BINDINGS:                         NIL
SI:FLAVOR-MESSAGE-HANDLER:                  NIL
SI:FLAVOR-NAME:                             TV:MULTIPLE-MENU-CHOOSE-MENU-MIXIN
SI:FLAVOR-LOCAL-INSTANCE-VARIABLES:         NIL
SI:FLAVOR-ALL-INSTANCE-VARIABLES:           NIL
SI:FLAVOR-METHOD-TABLE:                     ((MULTIPLE-CHOOSE NIL NIL ((METHOD TV:MULTIPLE-MENU-CH
SI:FLAVOR-DEPENDS-ON:                       (TV:MENU-HIGHLIGHTING-MIXIN)
SI:FLAVOR-DEPENDED-ON-BY:                   (TV:POP-UP-MULTIPLE-MENU-CHOOSE-MENU TV:MULTIPLE-MENU-
SI:FLAVOR-INCLUDES:                         NIL
SI:FLAVOR-PACKAGE:                          #<Package TV 3615215>
SI:FLAVOR-DEPENDS-ON-ALL:                   NIL
SI:FLAVOR-WHICH-OPERATIONS:                 NIL
SI:FLAVOR-GETTABLE-INSTANCE-VARIABLES:      NIL
SI:FLAVOR-SETTABLE-INSTANCE-VARIABLES:      NIL
SI:FLAVOR-INITABLE-INSTANCE-VARIABLES:      NIL
SI:FLAVOR-INIT-KEYWORDS:                    NIL
SI:FLAVOR-PLIST:                            (SI:MAPPING-TABLE-OFFSETS (TV:LAST-ITEM) SI:ORDERED-VA

                            Bottom of object
09/18/82 02:37:23 ROADS         USER:     Tyi___          ← F:>SYS>PRESS-FONTS>HELVETICA20B• 12
```

In the Inspector, the component objects are all *mouse-sensitive.* This means that if you point the mouse cursor at one of these components and click a mouse button, that component object gets inspected. It expands to fill the window and its components are shown. In this way, you can dive into a complex data structure, exploring the relationships between objects and the values of their components.

Components of the objects being inspected can be modified on the spot. This is done by using the menu at the top-right part of the screen to edit an object.

The Display Debugger

The Display Debugger provides a clear picture of the state of a Zetalisp process at the time of an error. It divides its area of the screen into seven panes (see figure 4):

- Display of the stack history with a pointer to the selected stack frame
- The names and values of the arguments to the selected stack frame
- The names and values of the local variables of the selected stack frame
- A command menu
- A Lisp interaction window
- An Inspector window
- An Inspector history list

You can select a different stack frame by clicking on it with the mouse; you can then examine its arguments and local variables.

The command menu in the Display Debugger includes commands to return an arbitrary value from any stack frame, to restart any function call in the stack, and to recover from the error and proceed if possible. The Display Debugger is interfaced to the Inspector so that you can inspect the various values you find in stack frames as well as the bodies of executing functions.

Since some programmers prefer to work with the mouse while others prefer not to use it, the Debugger supports both a keyboard-oriented style of working and a mouse-pointing style.

Figure 4. A Display Debugger window.

```
                                   More above
*EVAL
   232 BR-ATOM 236
   233 MOVE D-PDL LOCAL|3      ;ARGL
   234 (MISC) LENGTH D-PDL
   235 POP LOCAL|5             ;N-ARGS
   236 MOVE D-PDL LOCAL|1      ;FCTN
   237 (MISC) SYMBOLP D-IGNORE
   240 BR-NIL 245
   241 MOVE D-PDL LOCAL|1      ;FCTN
   242 (MISC) FSYMEVAL D-PDL
=> 243 POP LOCAL|1             ;FCTN
   244 BR 236
   245 MOVE D-PDL LOCAL|1      ;FCTN
   246 MOVEM LOCAL|2           ;CFCTN
                                   More below

   #<Stack-Frame *EVAL PC=243>

Args:                                  Locals:
Arg 0 (FORM): (TV:MENU)                Local 0 (ARGNUM): 0
                                       Local 1 (FCTN): TV:MENU
                                       Local 2 (CFCTN): NIL
                                       Local 3 (ARGL): NIL
                                       Local 4 (MAX-ARGS): NIL
                                       Local 5 (N-ARGS): 0
                                       Local 6 (ARG-DESC): NIL
                                       Local 7 (TEM): NIL
                                       Local 8 (COUNT): NIL
                                       Local 9 (QUOTE-STATUS): NIL
                                       Local 10 (REST-FLAG): NIL
                                       Local 11 (FEXPR-FLAG): NIL
                                       Local 12 (LAMBDA-LIST): NIL
                                 Bottom of stack
   (SI:LISP-TOP-LEVEL)
   (SI:LISP-TOP-LEVEL1 #<LISP-LISTENER Lisp Listener 1 11300240 exposed>)
  →(SI:*EVAL (TV:MENU))
   (DBG:FOOTHOLD)
   (DBG:SIGNAL-TRAP #<UNDEFINED-FUNCTION-TRAP 27204505>)
   (SIGNAL #<UNDEFINED-FUNCTION-TRAP 27204505>)
   (#<UNDEFINED-FUNCTION-TRAP 27204505>)
   ((METHOD CONDITION SIGNAL) SIGNAL T)
                                   More below
Return to normal debugger, staying in error context.
Supply a value to use as
Lisp Top Level in Lisp Listener 1

   What Error        Inspect        Return         Set arg          T
    Arglist           Edit           Throw          Search          NIL
>>Trap: The function TV:MENU is undefined.

_: Inspect item.  R: Get item into error handler.
09/18/82 03:15:05 ROADS          USER:     Tyi___
```

Peek: Examining the State of the System

The Peek utility program is a window-oriented interface that lets you examine and modify the state of the following:

- Processes
- Windows
- File system connections
- Virtual memory areas
- Network process status on the user's machine
- Network servers
- Network hosts
- Internal processor counters

All of the displays in Peek update themselves continually, so you can watch as processes change state, windows get created or killed, memory areas get allocated or deallocated, and so on (see figure 5).

Within Peek, menu commands can be selected with the mouse to perform operations on the following displayed items.

- Process
  Arrest, Unarrest, Flush, Reset, Kill, Debugger, Describe, Inspect.
- Window
  Expose, Deexpose, Select, Deselect, Deactivate, Kill, Bury.
- File system connection
  Reset, Describe, or Inspect a host-unit.
- Virtual memory areas
  Insert or Remove a display of all regions in an area.
- Network process status on the user's machine
  Show Hostat for selected host, Show Hostat for all hosts, Insert or Remove static Hostat, Supdup to selected host, Telnet to selected host, Send a message to a user on a selected host, Open or Close a network connection, Insert or Remove a display of packets on the receive or transmit list.
- Network Servers
  For each server: server name, process status, and file connections, each with its own menu of operations. All the operations listed under Network process status on the user's machine (above) can be applied to the server name. All the operations under Process (above) can be applied to the process.

Figure 5. A Peek window, showing the state of current windows.

```
Processes      Counters      Areas      File System      Windows
 Servers         Help         Quit        Hostat         Chaosnet
Screen Color

Screen Main Screen
  Peek
      Dynamic Mode Command   Deexpose
      Peek Pane 1            Expose
  Edit: REPORT.NEW SRC:<RO   Select
      Edit: REPORT.NEW SRC   Deselect
      Zmacs Mode Line Wind   Deactivate
          Typein Window 2    Kill
          Zwei With Typeou   Bury      le 2
  Lisp Listener 1
  FED (TR128)
      Fed 1
          Character Pane 1
          Unselectable Pane 1
          Basic Fed Pane 1
          Register Pane 1
          Command Menu Pane 3
          Highlighting Command Menu Pane 1
          Fed Status Pane 1
          Unselectable Chvv Pane 1
          Command Menu Pane 4
          Command Menu Pane 5
          Command Menu Pane 6
      Fed Mousable Typeout Window 1
  Inspect Frame 1
      Inspect Pane With Typeout 1
      Inspect Pane 1
      Inspect Pane 2
      Command Menu Pane 1
      Inspect History Pane With Margin Scrolling 1
      Interaction Pane 1
  Supdup -- not connected
  Main ZMail Window
      Zmail Summary Scroll Window 1
      Zmail Main Command Menu Pane 1
      Zmail Window 1
      Zmail Mouse Sensitive Mode Line Pane 1
          Typein Window 8
          Zwei With Typeout Unselectable 8
          Zwei Mini Buffer 8
  Telnet -- not connected
  Converse
      Converse
      Mode Line Window 3
          Typein Window 6
          Zwei With Typeout Unselectable 6
  Error Handler Frame 1
      Inspect Pane 3
      Inspect History Pane 1
      Args:
      Locals:
      Stack Scroll Pane 1
      Proceed Types Pane 1
      Command Menu Pane 2
      Error Handler Lisp Listener Pane 1


Windows
Expose the window.
09/18/82 03:37:51 ROADS        USER:      Menu Choose
```

The following operations can be applied to a file connection: Close, Insert Detail, Remove Detail, Describe, Inspect associated with.

The File System Editor: FSEdit

On the 3600, users manipulate the file system with a display-oriented tool called the File System Editor (FSEdit). This editor displays the name of every item in a directory. The item names are mouse-sensitive. Clicking with the mouse on a subdirectory name dynamically *opens* the subdirectory, displaying its contents. By opening and closing directories and subdirectories, users can poke around in the file system and see what is there. Users can also give a wildcard specification to select files for examination.

Once you have found a directory, file, or link on which to operate, you can click on it and get a menu of useful operations for that kind of item (see figure 6). The following operations are supported on directories:

- Delete or undelete (soft deletion)
- Expunge contents (hard deletion)
- Create new directories and links
- Rename
- Invoke the dumper to save the contents on a backup tape
- View and edit the directory properties

The following operations are supported on regular files:

- Delete or undelete
- View contents
- Edit
- Rename
- Invoke the dumper to save the contents on a backup tape
- View and edit the file properties
- Make a hardcopy of the file

The Font Editor: FED

The font editor program, called FED, allows users to create, modify, and extend type fonts that appear on the display screen (see figure 7). Fonts are drawn on a grid using the mouse or are edited on the gray plane, a mechanism for adding pieces of characters onto characters that are being built. The gray

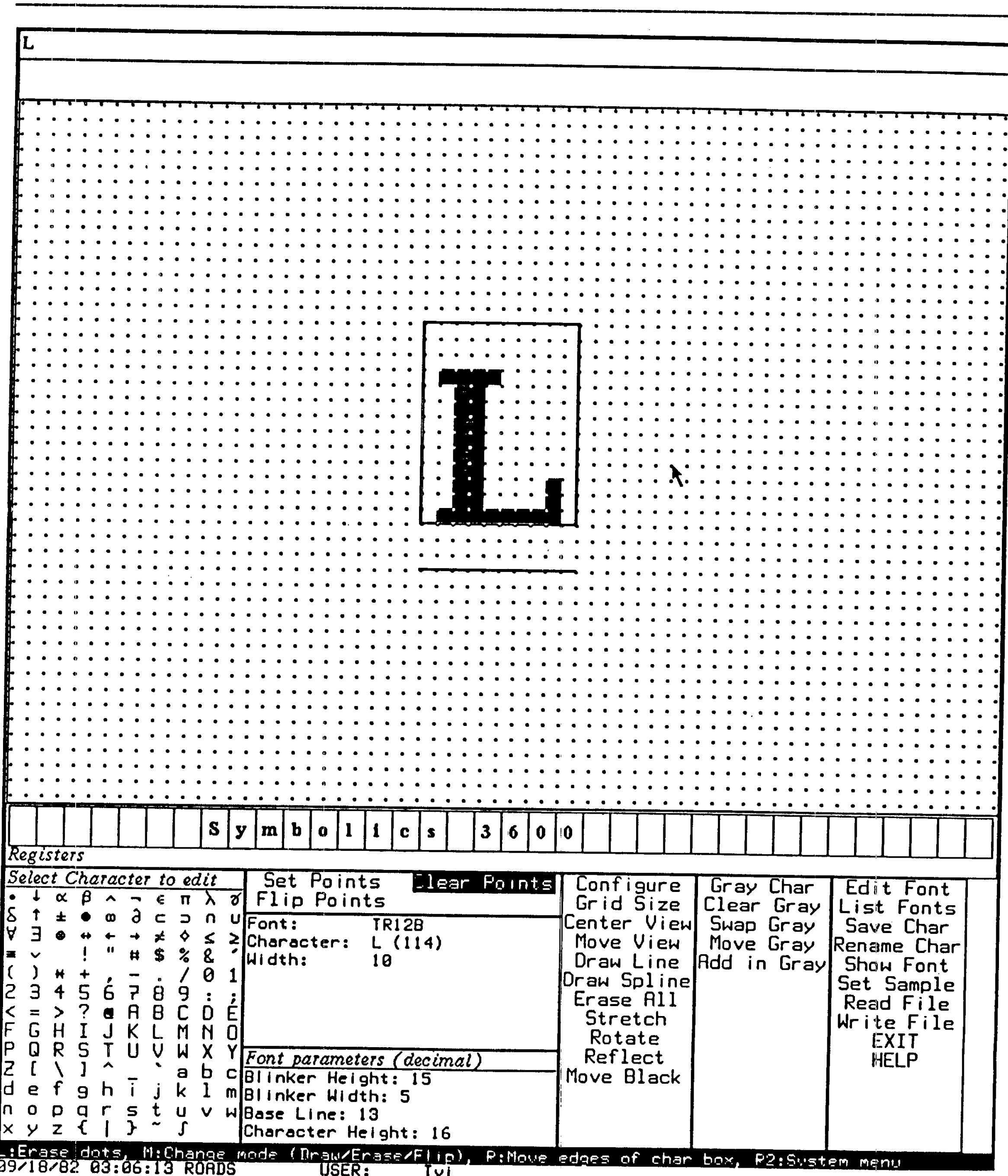
Figure 6. A File System Editor window, with a momentary menu of directory operations.

```
File System Operations
  Tree edit root          Tree edit any       Tree edit Homedir      Maintenance
  Incremental Dump        Complete Dump       Salvage                Reload/Retrieve
  Print Disk Label        Print Herald        Flush Free Buffer      Free records
  Server Shutdown         Server Errors       Lisp Window            Flush Typeout
  HELP                    QUIT

Directory operations: >sys
        Delete X
        Close
        Decache
        Expunge
   Create Inferior Directory
      View Properties            5725(16)    08/07/82 22:18:06 (09/08/82)  MMcM
      Edit Properties            9209(8)    $07/27/82 07:45:40 (10/20/82)  DLA
      New Property              11362(16)    08/27/82 09:59:17 (10/30/82)  Zippy
      Create link                5284(8)     04/09/82 17:38:13 (10/07/82)  BEE
      Rename                     4619(16)    04/09/82 17:38:32 (09/08/82)  BEE
      Wildcard Delete            4175(16)    08/07/82 22:20:04 (09/08/82)  MMcM
    Link Transparencies         14699(8)     10/17/82 20:52:02 (10/29/82)  CWH
   alarm.qbin.16          3      5349(16)    10/28/82 20:40:26 (10/30/82)  DLA
   CAFE.BIN.1             1      1398(16)    08/07/82 22:21:13 (09/08/82)  MMcM
   CAFE.LISP.7            2      5949(8)    $08/13/82 13:12:08 (10/20/82)  dlw
   CAFE.QBIN.7            2      1774(16)    08/14/82 11:02:23 (10/30/82)  DLA
   color-hacks.lisp.4     2      3585(8)     09/01/82 09:02:01 (10/07/82)  dlw
   COLXOR.BIN.1           3      4854(16)    08/07/82 22:24:14 (09/08/82)  MMcM
   COLXOR.LISP.56         4     13743(8)    $10/05/82 14:10:16 (10/20/82)  DLA
   COLXOR.QBIN.54         4      6426(16)    10/05/82 15:50:51 (10/30/82)  Zippy
   CROCK.BIN.4            2      3353(16)    08/11/82 04:52:56 (09/08/82)  Moon
   CROCK.LISP.7           3      7952(8)    $08/11/82 04:52:40 (10/20/82)  Moon
   CROCK.QBIN.10          4      6933(16)    08/27/82 09:57:56 (10/30/82)  Zippy
   DC.BIN.1               3      4926(16)    08/07/82 22:27:07 (09/08/82)  MMcM
   DC.LISP.5              3      8169(8)    $07/27/82 21:10:05 (10/20/82)  MMcM
   DC.QBIN.9              5      8407(16)    08/27/82 09:56:38 (10/30/82)  Zippy
   demo.LISP.37           2      4536(8)     08/12/82 13:09:54 (09/08/82)  dlw
   DEUTSC.BIN.1           2      1718(16)    08/07/82 22:28:47 (09/08/82)  MMcM
   DEUTSC.LISP.33         2      5117(8)    $02/13/82 01:31:46 (10/22/82)  MMcM
   DEUTSC.QBIN.34         2      2117(16)    07/24/82 04:35:09 (10/30/82)  DLA
   DLWHAK.BIN.1           3      4153(16)    08/07/82 22:29:27 (09/08/82)  MMcM
   DLWHAK.LISP.35         3      8987(8)    $07/24/82 08:06:17 (10/29/82)  DLA
   DLWHAK.QBIN.3          4      5881(16)    07/24/82 14:07:22 (10/30/82)  DLA
   DOCSCR.LISP.3          5     18853(8)     10/09/81 04:09:13 (10/08/82)  MMCM
   DOCTOR.LISP.8          2      6390(8)     10/09/81 04:09:05 (10/08/82)  MMCM
   edit.LISP.2            1       804(8)     08/12/82 12:58:40 (09/23/82)  dlw
   GEB.BIN.2              4      5615(16)    08/07/82 22:30:48 (09/08/82)  MMcM
   GEB.LISP.28            4     13543(8)    $09/27/82 10:46:24 (10/22/82)  dlw
   GEB.QBIN.29            5      7984(16)    09/30/82 13:13:29 (10/30/82)  DLA
   HAKDEF.BIN.2           2      2788(16)    08/07/82 22:14:14 (09/08/82)  MMcM
   HAKDEF.CROSS-QFASL.2   3      4064(16)    08/07/82 22:15:24 (09/08/82)  MMcM
   hakdef.lisp.15         2      3694(8)  !  10/29/82 15:14:56 (10/29/82)  DLA
   hakdef.qbin.11         3      4772(16) !  10/29/82 18:25:57 (10/30/82)  DLA
   HCEDIT.BIN.2           2      3416(16)    08/07/82 22:32:21 (09/08/82)  MMcM
   HCEDIT.LISP.30         3      9488(8)    $08/07/82 03:13:22 (10/20/82)  MMcM
   HCEDIT.QBIN.30         3      4277(16)    08/08/82 21:24:27 (10/30/82)  DLA
   INTRO.BASIC.1          1       753(8)     10/09/81 04:08:56 (10/08/82)  MMCM
   LISS.LISP.3            1      2373(8)     10/09/81 04:08:51 (10/08/82)  MMCM
   metr.lisp.3            5     16148(8)     08/13/82 14:38:35 (09/14/82)  lispm

File System Editor
Mark this directory as deleted.
10/31/82 23:35:02 ROADS        USER:      Menu Choose
```

Figure 7. The font editor screen.



plane acts as a "shadow" behind the drawing plane. It lets users compare the character under construction with another character.

With FED, characters of a font can be stretched or contracted and then examined in the context of a sample character string.

Tools for Managing Large Systems

As programs have grown larger in recent years, it has been recognized that a primary bottleneck in software development is *software complexity*. The job of a program-development system is to help the programmer manage that complexity. As this section explains, the Zetalisp environment provides a battery of language features and interactive tools to make working with large programs easier.

Defining a System: Selective Modification and Recompilation

Large Lisp programs are usually divided into several different files. This yields manageable pieces of code for text editing and compiling. Such a set of files can be presented to Zetalisp as a *system* and then managed as a unit. A *system declaration* lists all of the files in a system and specifies their interdependencies.

For example, one file might provide definitions that must be loaded into the Lisp environment in order for a second file to be compiled correctly. When asked to compile a system, the software automatically finds all of the files that have been modified since the last time they were compiled. The software then offers to compile them, first performing any actions that are required by the dependencies, such as loading definition files.

Recompilation can be done selectively. In this mode, you are asked whether the file should be compiled and are presented with a directory listing showing the existing versions of the file with their creation times and authors. You are offered a source-to-source comparison between the installed version and the latest version, or between any other versions. This allows you to examine what changes have been made to the software and to audit these changes before approving the recompilation.

Patching Bugs

The patch system for the 3600 is useful for correcting software errors (bugs) and for distributing software improvements and updates.

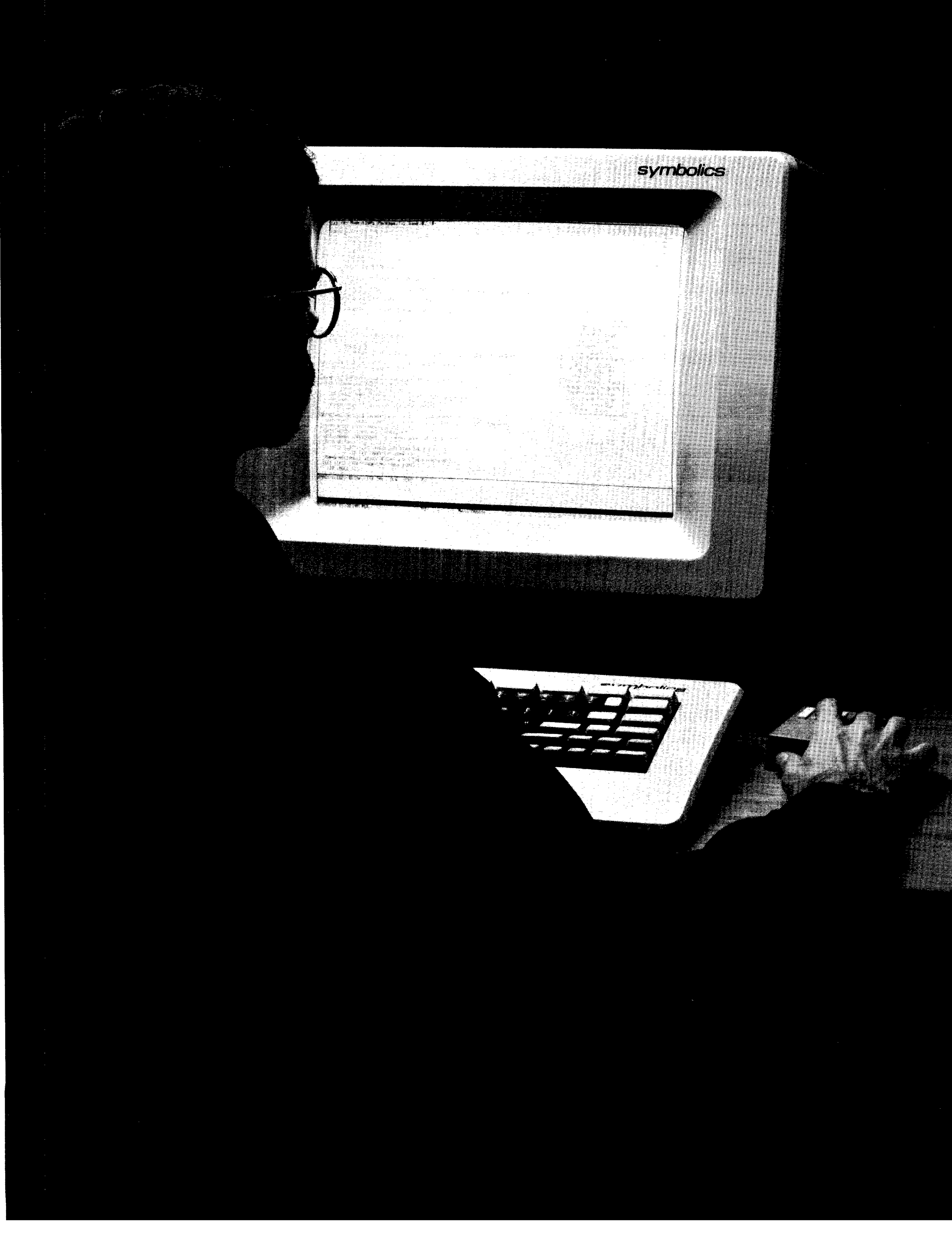
Software bugs are inevitable in large and complex software systems. A desirable maintenance goal is to fix the problem and quickly distribute the correction to all users of the system. If the system is a small program that is loaded into the Lisp environment by each user every time it is used, you just recompile the files and users will get the latest version when they load them. However, if the system is large and versions of it are stored away in saved Lisp environments, it is time-consuming to completely reload and recompile the system.

Zetalisp provides a facility whereby the maintainers of the system can create patch files containing function redefinitions or other Lisp forms that correct software problems and make these patches publicly available. Users can give a command that loads all of the latest patches to the system.

Simple commands in Zmacs (Add Patch and Finish Patch) make it easy to install new patches to a system. Changes are inserted in source form and are automatically recompiled.

The patch system performs a number of tasks to construct a documented patch:

- Assigns unique two-part version numbers based on the version of the program that was loaded and the level of patches that have been installed.
- Includes author of patch.
- Includes reason for patch.
- Documents environment in which patch was constructed.


symbolics

3600 Software: Languages

This chapter provides summaries of the language systems on the 3600, including Zetalisp, Symbolics Common Lisp, the Interlisp Compatibility Package, FORTRAN 77, LIL, and MACSYMA.

The Zetalisp Language

Zetalisp, a powerful modern dialect of Lisp, is the principal language of the 3600. All system software is written in this dialect.

Zetalisp is a large language. An important advantage is its extensibility. This is in contrast to some languages which have a fixed set of language constructs. The extensibility of Zetalisp derives from its efficient, interactive, incrementally compiled, and dynamically linked programming environment. The environment is full of facilities that can be invoked with simple Lisp function calls or message-passing operations.

Data Types

Support for a range of data types is provided both in software and in hardware. With the tagged architecture of the 3600, run-time checking of data types is automatically performed in hardware for all operations. Type mismatches always signal an error. For example, obvious errors such as trying to take the *car* (first element) of a number or add to a nonnumber are always detected, even in compiled code. Similarly, attempts to reference an array outside its bounds also cause run-time errors.

The Zetalisp language provides many predefined data types including:

- Numeric
  Includes fixed-point, IEEE-standard floating-point, and infinite-precision fixed-point (*bignum*) numbers. Bignums can represent millions of digits.
- Multidimensional array
  Can contain a mixture of Lisp objects of any type, including arrays. Special types of arrays exist to store fixed-point numbers, characters, or bits in a packed format. Simple Lisp functions make arrays and examine and alter their elements. Array bounds can be adjusted dynamically.
- Character string

Contains characters. Supported as a type of array, so the regular array accessing functions can be used to manipulate the characters of a string directly. In addition, an extensive set of string-manipulation functions provide such capabilities as string searching, string concatenation, substring extraction, string reversal, and string trimming.

↗ Plane
Contains Lisp objects. Special multidimensional array data type whose bounds, in each dimension, are plus-infinity and minus-infinity. All integers are valid as indices.

↗ Structure
Contains named components, each of which can be any Lisp object. The user can specify the type identifier for such structures and also control their printed representation.

↗ Flavor
Contains data (instance variables) and operations on the data (methods). Provides support for object-oriented programming based on message-passing. (See page 38.)

Zetalisp also allows users to define their own types.

Program Control Mechanisms

Having a wide range of program control mechanisms is important to managing the flow of control in complex programs. Zetalisp has a comprehensive set of control constructs, including conditionals, dispatch forms, a powerful loop facility, safe and structured nonlocal exits, coroutines, and processes.

Standard Control Structures

Zetalisp has all of the control structures employed by conventional languages. It has simple conditionals, multiple branching conditionals, and several kinds of dispatching (**case**) constructs. Several iteration constructs are provided, including **dolist** and **dotimes** for simple iteration over a list or a sequence of numbers. The Maclisp **do** is available for more general iteration, allowing users to step many variables in parallel and perform arbitrary end-tests. The traditional Lisp **prog** feature with **go** is also supported in Zetalisp, although it is rarely needed.

The Loop Facility

The powerful **loop** macro of Lisp is a programmable iteration facility. Loop forms are intended to look like stylized English rather than traditional Lisp code. Loops consist of three parts:

- Initialization
- Body
- Exit

Each part is introduced by special keywords. Over 40 loop keywords are available, including **with, for, repeat, initially, finally, collect, do, append, count, sum, always, never, if, thereis, unless, when,** and **until**. Programmers can combine these into very flexible iteration expressions.

Advanced Control Structures

Zetalisp provides advanced control structures. One of these is the catch/throw mechanism for structured nonlocal exits. If, during evaluation, a **throw** form is encountered, the system looks for a **catch** form with a tag matching that of the **throw**. If found, the body of the **catch** form is terminated, and **catch** returns the values given to **throw**. One of the primary uses of this mechanism is to exit a program when an error is detected. It can also be used, for example, to get out of levels of looping and recursion upon finding an item for which the program has been searching.

Zetalisp also provides the **unwind-protect** special form which is used to ensure that certain "clean-up" computations occur, even in the context of a nonlocal exit (for example, a catch/throw exit). By setting up unwind-protect forms around code, you can arrange for temporary effects to be undone whenever an evaluation finishes, whether normally or by means of a nonlocal exit.

Coroutine Structures

Zetalisp provides two kinds of coroutine control structures. Simple coroutines can be created using *closures*. A closure is a Zetalisp function along with some saved variable-binding information. A function can be written as a generator that saves its state in some variables. Then creating a closure of that

function over those variables gives a new instance of the generator.

More powerful and general coroutines can be created using *stack groups*. A stack group holds the entire state of an executing Lisp program, including the control stack history and the state of the bound variables. At any time, one stack group is the currently executing stack group. It can call a second stack group; this transmits a value to the second stack group and starts its computation running. Multiprocessing, implemented using stack groups and a scheduler, is also provided (see page 66).

Function Calling

Zetalisp provides flexible argument-passing mechanisms for functions. Features include:

- Any number of parameters to functions
- Optional parameters
- Arguments to functions passed as keywords

Multiple Values

Zetalisp allows a function to return any number of values. This is useful when a function computes more than one interesting value. In ordinary Lisp dialects, you would have to create a data structure (such as a list) to contain the values, and return it; the caller would then have to take this apart again. In Zetalisp, the caller can specify variables that should be bound or set to the returned values.

Input/Output Mechanisms

Input from and output to a number of places simultaneously is handled via the streams facility in Lisp. Many of the software facilities for input and output are based on generic operations (see the discussion of flavors, page 38). The character set, streams, and formatted output are discussed in this section.

Zetalisp Character Set

Zetalisp represents characters as fixnums (small integers). Fundamental characters are represented in eight bits. In some contexts, characters are represented in 16 bits to allow font information to be attached to the characters.

Streams

Input and output in Zetalisp are processed by sending messages to *stream* objects. A stream is a Zetalisp object that can act as a source or sink of sequential data. An input stream object is sent messages such as:

- Read the next datum
- Read a line
- Are data available?
- Clear buffered input

An output stream object is sent messages such as:

- Write this datum
- Write this string
- Wait for buffered output to finish going to device

Predefined streams are provided for communication with the keyboard, windows on the screen, files in any accessible file system, network connections, editor buffers, and character strings.

All input functions work on any input stream, and all output functions work on any output stream. Streams can be unidirectional or bidirectional. Streams can be interconnected into arbitrary patterns. Using a system of predefined flavors, programmers can define stream objects to create specialized input/output interfaces.

Formatted Output

Zetalisp provides an output function called **format** for merging literal and computed strings. The string can contain escape sequences to specify the following operations:

- Print numbers in any base, with specified field widths and padding characters
- Print English cardinal and ordinal numbers
- Print Roman numerals
- Print pluralized words
- Space to particular columns
- Output control sequences
- Conditionalize text
- Iterate over lists to justify fields and perform other actions

Formatted output can be sent to a character string or to any output stream.

Predefined Functions

Literally thousands of functions and flavors are predefined in Zetalisp. For example, to sort a list or an array, the **sort** function is used, which provides a carefully optimized quicksort. If a hash table is needed, the **make-hash-table** function is used, and then simple Lisp functions can be used to insert elements and look them up. Pseudorandom numbers are available by calling the **random** function. It is possible to convert dates and times between various formats, and print them out or read them in, by calling predefined functions.

Zetalisp provides a range of numerical functions, including basic arithmetic operators, transcendental functions, numerical type conversions, logical operators (including bit and byte manipulation primitives), IEEE floating-point arithmetic, infinite-precision fixed-point arithmetic, complex numbers, and rational numbers.

More advanced numerical operations are also part of the standard Lisp environment. For example, two high-level functions are used to solve systems of linear equations. Matrix multiplications, inversions, transpositions, and determinants can be invoked with a single function call each. These are all part of the Lisp environment and are available at any time.

Packages for Independent Namespaces

The Lisp environment is based on a single, large workspace which contains all of the functions and global variables of all of the programs in the system. To prevent naming conflicts between different programs that both happen to choose the same name for some function or variable, the system provides a hierarchical system of name spaces, or *packages*, for different programs.

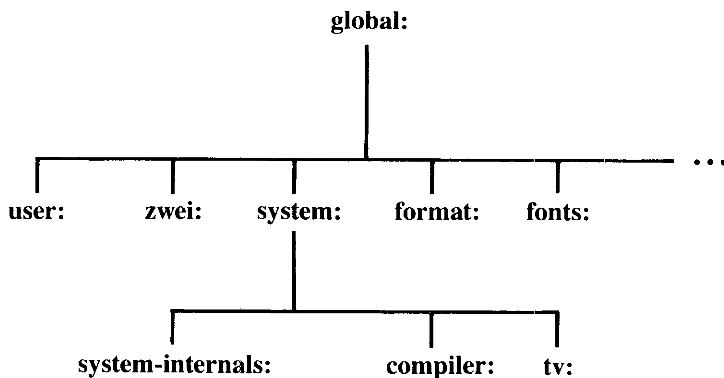
A package is an association between character strings and Lisp symbols. In different packages, the same character string can be associated with different symbols. For example, **si:foo** and **tv:foo** refer to distinct symbols named **foo**.

The package prefix does not always have to be specified

explicitly, since a *current package* is always used by default. In Zmacs, programmers can set the attribute list of a file so that a package name is always associated with that file.

When a file without an explicit package declaration is loaded into a Lisp Listener, all symbols in the file are associated with the current package. Individual symbols can be prefixed with another package name that overrides the default.

Figure 8. A package hierarchy.



Packages are arranged in a strict hierarchy. A subpackage can inherit symbols from its parent package. In figure 8, the package **compiler:** inherits symbols from its parent package **system:**, which in turn inherits from its parent package **global:**.

Interactive Debugging Tools

Several interactive facilities help the programmer examine and debug programs.

- The trace facility provides a means of finding the callers of a particular function or set of functions.
- **apropos** finds all of the symbols whose name contains a given substring; this is useful if you remember only part of a name.
- Memory regions can be marked so that an error can be signalled when a program incorrectly writes into the wrong memory region.
- Single-stepping through a program is supported, allowing you to watch each evaluation as it happens and examine the environment between steps.

Flavor System: Language features to support object-oriented programming style

Symbolics provides support for object-oriented programming through a collection of language features known as the Flavor System. Object-oriented programming deals with objects, which are instances of types, and generic operations defined on those types.

In object-oriented programming, you define a type by defining the data known to the type and the operations that are valid for those data. Then you create instances of the type. Each instance maintains a local state and has an interface to the world through the defined operations.

Thus, in object-oriented programming, data and procedures are encapsulated within an instance of the type. This shields users of a facility from the details of its implementation, resulting in programs that are easier to describe, develop, and maintain.

What is the Flavor System?

The Flavor System is Symbolics implementation of the language features that support object-oriented programming. *Flavors* are the abstract types; *methods* are the generic operators. The objects are *flavor instances* that you manipulate by sending *messages*, which are requests for specific operations.

The Flavor System, you define the flavors and the methods associated with them in one part of a program. Then, in another part of the program, you create instances of the flavors and send them messages to request that operations be performed.

The Flavor System provides a unique and powerful mechanism for declaring the relationships between flavors. A new flavor definition can be built from component flavors; the method definitions for each flavor can override, augment, or modify the methods from the component flavors.

The flavor dependencies form a graph structure; they are not constrained to be hierarchical as in some languages that support an object-oriented style. Thus, the Flavor System provides the means for building arbitrarily complex flavors while retaining the advantages of modularity and maintainability.

Figure 9 contains a graphical representation of flavor combination and of the process of instantiating a flavor.

Using Flavors

The Flavor System provides a constructor for flavors. Allocating and using a flavor instance is a matter of calling the constructor function for the required flavor. This process is known as *instantiation.* For example, suppose you need an instance of a flavor called system-process:

```
(setq p3 (make-instance 'system-process))
```

The definition for the flavor can contain declarations of initial values for the local variables and, in many cases, you can also declare initial values in the call to the constructor.

Message passing

Once a flavor object has been instantiated (as in **p3** from the example above), it can receive messages. The messages can be accessors for local state variables or they can be requests for the results of procedures. For example, suppose that one of the local variables for a system process records its priority. The instance returns the value of this variable when it receives the appropriate accessor message:

```
(send p3 ':priority)
```

Again, the accessor messages can be created automatically by the Flavor System.

Similarly for operations on a process, you send a message to the process instance requesting that the operation be performed. For example, suppose that one of the operations defined for a system process is that it can be reset:

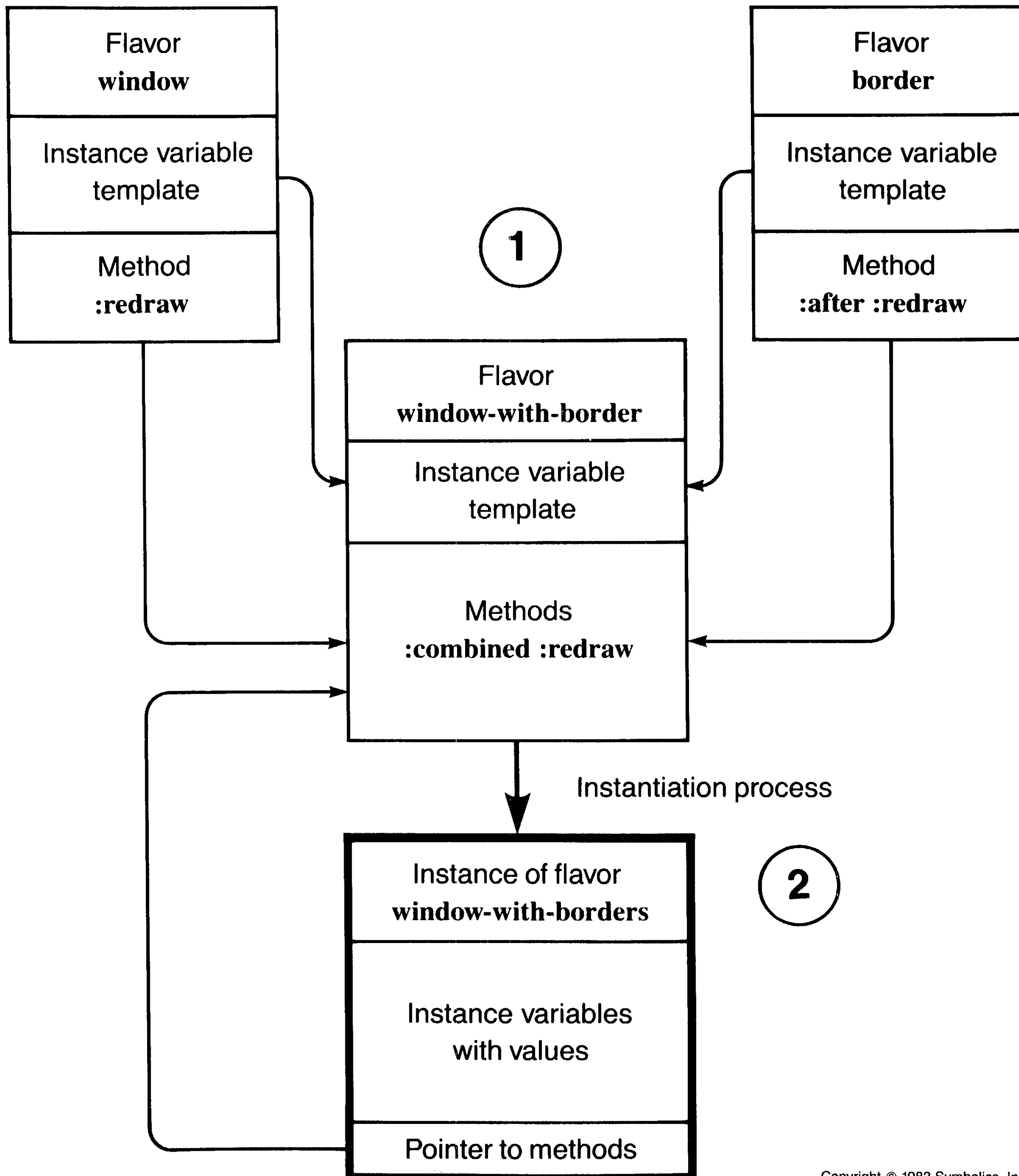
```
(send p3 ':reset)
```

Of course, the operations are implemented as function calls and can accept arguments. Suppose that to reset a process you had to supply a validation code that was obtained from an object of another flavor. The call might look like this:

```
(send p3 ':reset (send last-error ':reason))
```

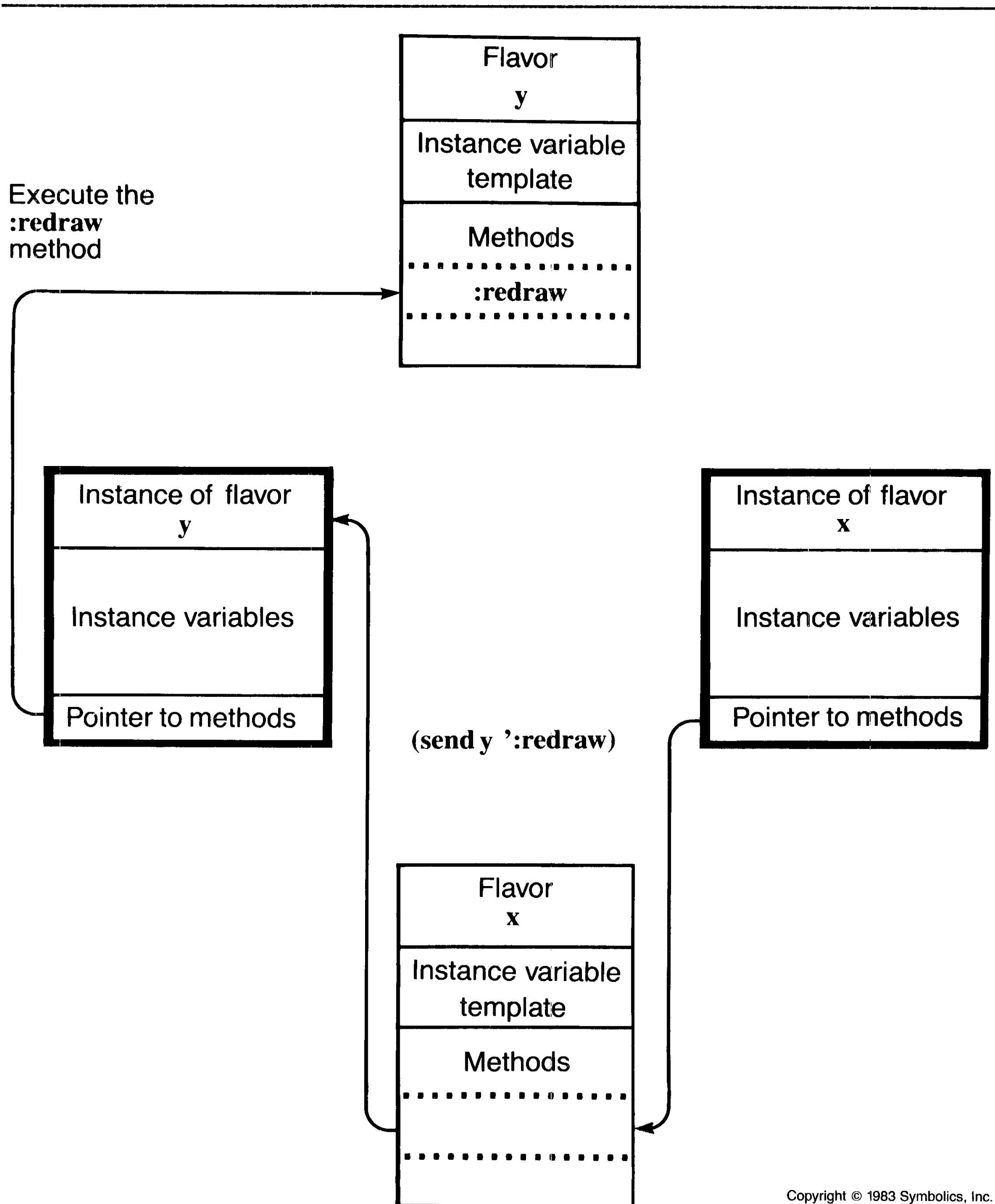
Figure 10 contains a graphical representation of the message-passing operation.

Figure 9. Flavor combination (1) and instantiation (2).



Copyright © 1983 Symbolics, Inc.

Figure 10. Message-passing operation. Flavor instance **x** sends a message to flavor instance **y**.



Copyright © 1983 Symbolics, Inc.

Designing Flavors

Flavor definitions include several kinds of information:

- Names of other flavors from which the new one inherits local variables and methods.
- Identifiers for local variables, called *instance variables*, and any procedures or values for initializing these.
- Methods for implementing the operations required for objects of this flavor.
- Declarations of relationships and interdependencies with other methods and flavors.

Component flavors

The following example creates a new flavor, called **new-mouse-sensitive-window**, that inherits the instance variables and methods from four flavors in the **tv:** package:

```
(defflavor new-mouse-sensitive-window ()
  (tv:basic-mouse-sensitive-items
   tv:borders-mixin
   tv:top-box-label-mixin
   tv:window))
```

Some flavors are designated by convention as *base flavors*, serving as the foundation for building a flavor family. Other flavors are designated as *mixin flavors*, serving to implement particular features that would be required for other flavors. User-defined flavors are constructed by combining these base and mixin flavors into the exact flavors required for the application.

Complex flavor combinations can be defined. Figure 11 shows the relationships among the various base and mixin flavors specified in the definition of the flavor **tv:menu**. In this kind of nonhierarchical structure, the flavors being inherited can overlap. The Flavor System takes care to eliminate redundancy by including a subflavor the first time it appears in a depth-first traversal of the structure but ignoring it on subsequent occurrences.

The Flavor System allows you to specify interdependencies among flavor definitions. Certain flavors can be specified as required for the definition of another. Then, if the required flavors are not included as a result of the inheritance mechanism, they are added explicitly.

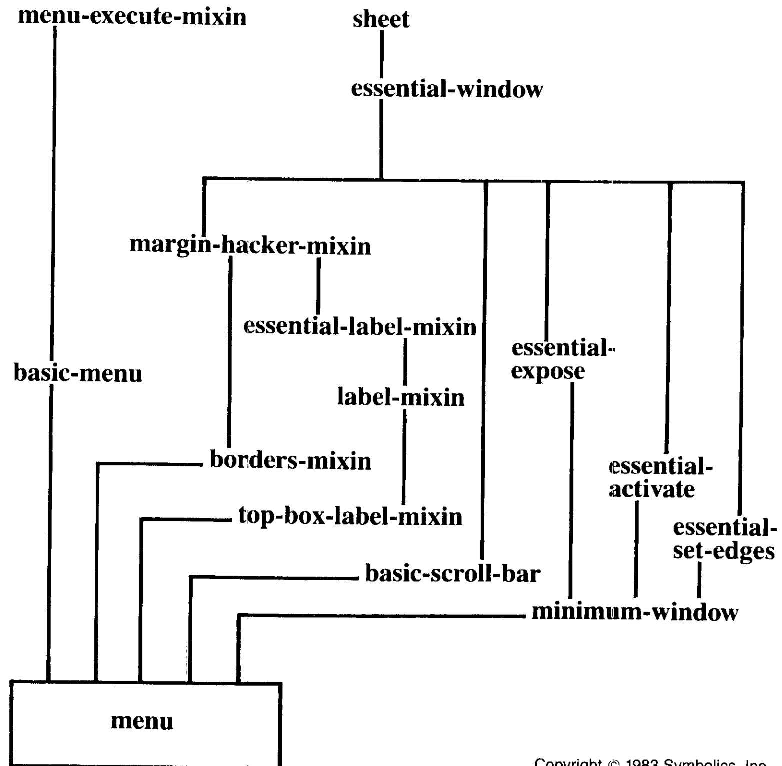


Figure 11. The inheritance network of the flavor **tv:menu** built by combining many component flavors.

Instance variables

The instance variables hold local state information for an instance. The Flavor System provides time-saving mechanisms for managing instance variables. Initial values can be assigned automatically as part of the process of creating the instance or can be supplied explicitly by the user as part of the call to the constructor that creates the instance.

The instance variables can be declared as internal or external to the instance. The Flavor System automatically provides accessor methods for instance variables whose values are available externally and mutator methods for instance variables that can be changed by other parts of the program.

Methods

Methods are function definitions that implement the operations defined for each flavor. Methods can be defined directly for a flavor or can be inherited from component flavors. Since inherited methods might well conflict with requirements for a flavor being defined, the Flavor System provides a means for specifying how to create a combined method that combines all of the inherited methods specified in component flavors.

The Flavor System provides mechanisms that allow methods to be combined in different ways. One method can completely override an inherited method. The order in which methods of the same name are performed can be declared. An inherited method can also be executed conditionally, depending on the flavor declaration.

Signalling and Handling Conditions

The condition system provides the mechanism for detecting and responding to exceptional events that occur during execution.

An *event* is "something that happens" during execution of a program. It is some circumstance that the system can detect, like the effect of dividing by zero. Some events are errors — which means that whatever happened was not part of the contract of a given function — and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.

The reporting process is called *signalling* and subsequent processing is called *handling.* A *handler* is a piece of user-supplied code that assumes control when it is invoked as a result of signalling. The 3600 software includes default mechanisms to handle a standard set of events automatically.

The condition system for reporting the occurrence of an event is implemented using flavors. Each standard class of events has a corresponding flavor called a *condition.* For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**.

The mechanism for reporting the occurrence of an event is called

signalling a condition. The signalling mechanism creates a *condition object* of the appropriate flavor. The condition object is an instance of the flavor. The instance variables for the condition object contain further information about the event, such as a textual message to report and various parameters of the condition.

Handlers are pieces of user or system code that are applied to a particular condition. When an event occurs, the signalling mechanism searches all of the currently available handlers to find the one that corresponds to the condition. The handler then accesses the instance variables of the condition object to find out more about the condition.

The condition system provides flexible mechanisms for determining what to do after a handler runs. The handler can try to *proceed,* which means that the program might be able to continue execution past the point at which the condition was signalled, possibly after a user repairs the problem. Any program can designate *restart points.* This facility allows a user to retry an operation from some earlier point in a program.

Inheritance mechanisms in the Flavor System allow you to design conditions that are very specific to a particular set of circumstances and others that are more general. Groups of similar events can be handled in a consistent way by a related family of condition flavors. For example, **fs:file-operation-failure** inherits behavior from **fs:file-error**. You choose the level of condition that is appropriate for a program to handle according to the needs of a particular application.

Macros: Extending the Language

Most languages, have a fixed set of syntactic constructs. In Zetalisp, however, you can add constructs to suit particular needs or tastes, or to tune the language to a particular application, by creating a Lisp macro. Lisp provides tools to make writing macros easy and convenient.

Macros in Zetalisp are different from macros in traditional languages — rather than manipulating text, they manipulate the

structure of the program. A Lisp program is made of Lisp data structure, with lists, symbols, numbers, and so on. A Lisp macro is a function that manipulates the structure of a program. It translates the syntax that programmers invent for their extensions into existing Lisp constructs.

When building a large software system in Lisp, programmers typically use macros to create language extensions that have specific features useful for their system. Then they write the application in this extended language. Specialized languages have been built on Zetalisp for such diverse purposes as computer graphics, digital signal processing, expert problem solving, and VLSI circuit design.

Access to System Subprimitives

Zetalisp provides unrestricted access to the lowest levels of the software system and the processor. A special set of functions, called subprimitives, can be used to manipulate internal data structures and deal with levels lower than the Lisp language itself. Many of the fundamental functions and facilities of Lisp are written as Lisp programs that call these subprimitives. While it is rare that user programs need to use these facilities, they are accessible.

Common Lisp Compatibility

A subset of Lisp, called Common Lisp, is being defined by a joint industry/university project group. This group consists of designers of several existing Lisp dialects.

The goal of the Common Lisp designers is to provide portability between different Lisp dialects of Maclisp descent. Any program written entirely in Common Lisp runs correctly in any of these dialects. Common Lisp is intended to be very stable; only well-tested extensions will be accepted into Common Lisp.

Symbolics is strongly committed to supporting this standard. Common Lisp is a proper subset of Symbolics Common Lisp.

Although the Common Lisp standard allows programmers to declare data types at compile-time, this is only for the benefit of software running on traditional computers. The 3600's run-time data-type checking in hardware makes this practice unnecessary.

Symbolics Common Lisp will be supported on the 3600 after the definition of Common Lisp is frozen. All Zetalisp features will be preserved as new features are added.

Other Supported Languages and Systems

In addition to the native Zetalisp language, Symbolics supports Interlisp and FORTRAN 77. Programs written in these languages can be executed on the 3600.

LIL (Lisp-like Implementation Language) is the systems programming language for the front-end processor (FEP)[2] and the 3600 console. Interrupt service routines and other input/output programs are written in LIL.

MACSYMA, a symbolic mathematical system, is also available for the 3600.

Interlisp Compatibility Package

The Interlisp Compatibility Package is part of the 3600 systems software. It allows Interlisp-10, Interlisp-VAX, and Interlisp-D programs to run in the Zetalisp environment. The package provides tools for intermixing Interlisp and Zetalisp code and for incrementally converting Interlisp code into Zetalisp code. This means that as Interlisp programmers adjust their programs, they will be able to take advantage of the advanced features and capabilities of the 3600 and its extensive Zetalisp environment.

The Interlisp Compatibility Package includes tools for automating the code-conversion process. Programmers first convert an Interlisp source file into a file that can be loaded into the Zetalisp environment. They then load a group of functions that implement Interlisp operations. In this way, the Interlisp Compatibility Package maintains the functionality and appearance of the original Interlisp program, while allowing it to run in the 3600's environment.

The Interlisp Compatibility Package includes support for the following Interlisp features.

---

[2]The FEP is an MC68000-based computer system between the 3600 and the MULTIBUS. (See page 105.)

- Low-level pure Lisp functions including atom, list, and string manipulation
- Complete array manipulation capabilities
- **LAMBDA, NLAMBDA, PROG,** and other special forms
- Semantics for **Blocks,** especially **Specvars** and **Localvars** compiler declarations
- The pattern-match compiler
- All types of macros
- Such **Clisp** constructs as the iterative statement operators (including user extensibility), **Changetran, If-then-else,** and **Printout**
- All of the **Record** package
- The **Lambdatran** package

The 3600 provides equivalent functionality for many Interlisp features, including the window system, readtables, input/output streams, network tools, the Break package, and dribble files.[3]

Comparable facilities are provided on the 3600 for some of the Interlisp features that are not directly supported by the compatibility package. The functionality of the Interlisp structure editor, Programmer's Assistant, and Masterscope is realized on the 3600 by the Zmacs editor, which "knows about" Lisp syntax. **FIXSPELL** and **ERRORX** from Interlisp are replaced by the error detection and correction tools of the Display Debugger and by the Inspector. Interlisp's spaghetti stack and stack-manipulation functions are supplanted by Zetalisp's stack groups and flavors.

The **Clisp** infix notation of Interlisp is not supported. **LISPX** and **EVAL-QUOTE** notation from Interlisp are replaced by editing and reevaluation of expressions in the Zetalisp Rubout Handler. Upper/lower case independence in Interlisp is supported in Zetalisp by retaining lower case representation for any specified character or string.

---

[3]Unsupported features include the **DWIM** system, terminal tables, and the Interlisp File Package.

FORTRAN 77 Tool Kit

The FORTRAN 77 Tool Kit allows programmers to develop, debug, and run FORTRAN 77 programs on the 3600. It implements the full ANSI FORTRAN 77 Standard (FORTRAN X3.9-1978). It supports earlier dialects to the extent that they are compatible with FORTRAN 77. The Symbolics 3600 FORTRAN implementation integrates editing and source-level debugging facilities, providing FORTRAN programmers with the supportive development and maintenance environment of the 3600.

FORTRAN on the 3600 is different from FORTRAN on any conventional computer, personal or mainframe. Some of these differences are inherent in the Lisp Machine, for example, data-type checking. Others, like incremental compiling, provide for greater programmer productivity than is available from other systems.

The Tool Kit consists of:

- A compiler for the full FORTRAN 77 language, together with extensions of particular use to 3600 programmers.
- Extensions to Zmacs, the standard 3600 text editor, to support editing of FORTRAN programs.
- A Lisp-compatible run-time library, permitting full access to the 3600's input/output facilities, including access to files on other hosts via network connection.
- Compatibility with the Display Debugger, permitting debugging of FORTRAN 77 source code.

One characteristic of the Lisp Machine environment is that programs can be compiled and executed in an editor buffer. This facilitates *incremental compilation* — the process of compiling selected routines of a large system — which greatly speeds up the program-development process. The FORTRAN 77 Tool Kit makes this feature available to FORTRAN programs. It also brings the benefits of interactive programming to FORTRAN programmers. On the 3600 they can test small functions thoroughly without waiting for lengthy relinking.

The FORTRAN 77 compiler produces Lisp code, which is in turn compiled by the 3600's standard Lisp compiler. The Lisp

code produced by the compiler is an intermediate object language, incidental to producing machine code. All debugging is at the FORTRAN source-code level. Thus, a FORTRAN programmer need not think in terms of Lisp code while dealing with a FORTRAN program.

After compiling, FORTRAN routines are available for execution. They do not have to be linked and loaded as in conventional FORTRAN systems. Making small changes to large FORTRAN programs is therefore easier and faster.

Because they are compiled into Lisp code, FORTRAN programs become Lisp functions in the Zetalisp environment and are invoked with Lisp function calls. Calls in the FORTRAN program to FORTRAN library subroutines are actually calls to predefined Lisp system functions. FORTRAN programs can use any Lisp functions that have an appropriate calling interface. This compatible intermingling provides great flexibility, allowing a programmer to implement a particular routine in either Lisp or FORTRAN, whichever is more suitable to the specific situation.

The FORTRAN 77 package provides strong hardware data-type checking between logical, integer, and real data. This reduces data-equivalencing errors. This powerful feature uncovers bugs caused by uninitialized data which would remain hidden in other FORTRAN implementations.

All integers used in FORTRAN 77 are immune to overflow, in that infinite-precision integers (*bignums*, see page 31) are supported. In concert with infinite-precision integers, the FORTRAN 77 Tool Kit supports formatted output of large integers. Format specifications such as "I200" are meaningful.

LIL

LIL is the Lisp-like Implementation Language developed by Symbolics. It is designed to run on the MC68000-based front-end processor (FEP) and other processors associated with Lisp Machine systems. LIL is an implementation language in that its semantics are a close match to the macroinstruction sets of conventional processors like the MC68000. A typical application of LIL is the writing of the FEP's interrupt handlers for MULTIBUS devices. (See page 105.)

Like the programming language Pascal, LIL is strongly typed. This means that the type of each form must be known to the compiler. This type may not change while the program is running. LIL has user-defined types but no user-defined generic operations. Following is a type-definition example:

```
(deftype str-type
         (structure ()
          (next code-symbol)
          (kind symbol-kind)
          (minimum-loc word)
          (maximum-loc word)))
```

The syntax of a LIL source program is a list. The LIL compiler reads the source using the Lisp reader. Thus each LIL declaration, statement, or expression is one of two things:

- Atom
  A symbol or a constant
- Form
  A parenthesized list of atoms or forms

This requirement ensures that the syntax is similar to the parenthesis notation of Lisp. In LIL, however, symbols, atoms, and lists are syntactic units only. They are not a part of LIL's semantics. For example, LIL has no operations for performing computations on these entities. Like Lisp, LIL is an *expression language*. Most LIL forms return values that can be used in expressions.

Variable references in LIL are lexically scoped; function and type declarations must be at the top lexical level. In order to give programmers access to the underlying machine, LIL program units, such as psects, modules, and global variables, can be tied to physical addresses.

Lisp functions and a Zmacs editor interface are provided as support for LIL programmers. The standard **defsystem** and **make-system** functions support LIL code. In addition, "assembler-like" output listings are available as output options through both the Lisp and the Zmacs interfaces. The LIL compiler produces code that is routed directly to the linker. From the linker, code can be sent to relocatable output files, to a

PROM (programmable-read-only-memory) burner, or to the front-end processor (FEP).

MACSYMA

MACSYMA for the 3600 is a large, highly sophisticated software system for symbolic mathematical manipulation. The system handles symbolic mathematical expressions, following the rules of algebra and calculus, and produces symbolic closed-form solutions. MACSYMA serves as an expert mathematics assistant to applied mathematicians, scientists, engineers, and educators.

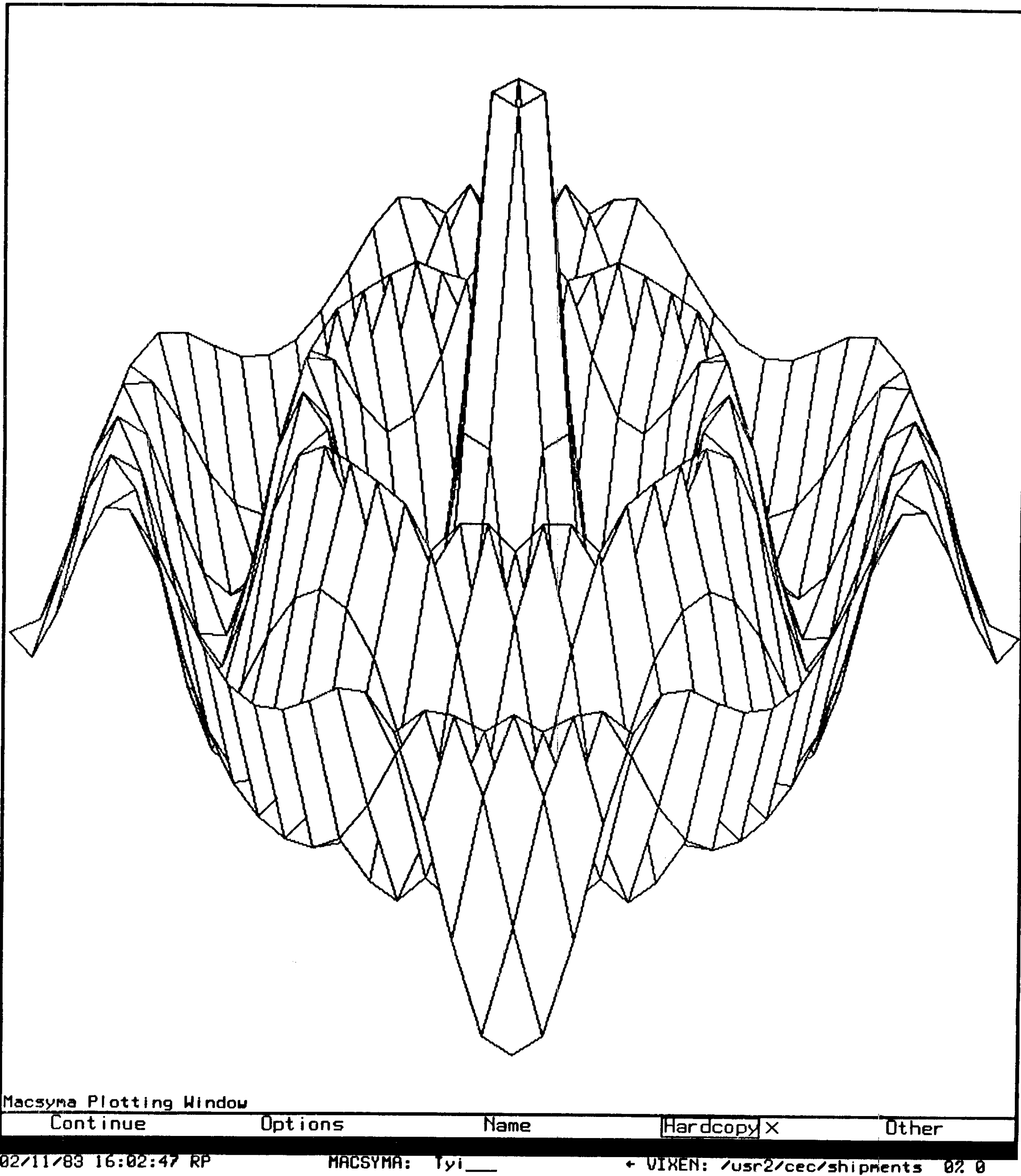
The implementation of MACSYMA on the 3600 represents the culmination of a long software development effort that began in the early 1960s. Over 100 programmer-years of software design and implementation effort have produced a comprehensive package with a range of capabilities, including:

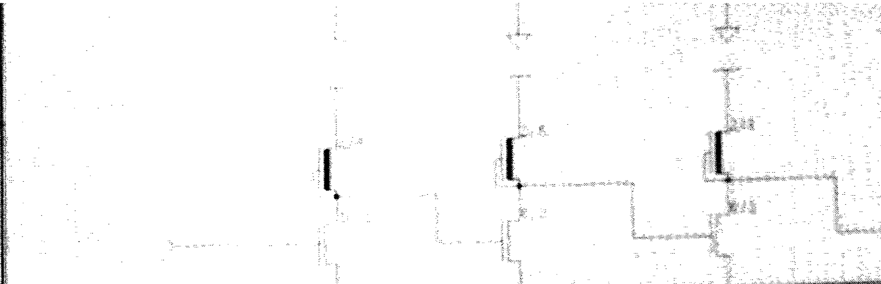
- Symbolic integration
- Symbolic differentiation
- Linear or polynomial equation-solving
- Algebraic simplification and factoring
- Expansion of Laurent and Taylor series
- Solution of differential equations
- Computation of Poisson series
- Tensor and matrix manipulation
- 3-dimensional curve plotting

MACSYMA on the 3600 has three main advantages over earlier implementations.

- Users work within a 256 Mword (1 Gbyte) virtual address space. This removes a traditional frustration — inadequate memory — for MACSYMA users.
- The entire machine is made available to the user. With no timesharing, interaction with MACSYMA is fast.
- On the 3600, users can take advantage of the fast, high-resolution display to generate detailed visual graphs of symbolic functions (see figure 12).

Figure 12. A MACSYMA window in use, example of the **PLOT3D** function.




PAN
REDISPLAY
VIEW-MENU
NAME
PROPERTIES
CLEAN
TOGGLE-LAB
EXIT
REDISPLAY
CONFIGS
COLUMN
SPICE-SIMUL
DUMP
LOAD
LGP
Windows
This window
Programs
Create
Attributes
Lisp
Select
Refresh
Edit
Split Screen
Bury
Inspect
Layouts
Kill
Mail
Edit Screen
Reset
Font Edit
Set Mouse Screen
Arrest
Trace
Un-Arrest
Emergency Break
Hardcopy
File System
NEW-SCHEMA:
Console idle 22 minutes

symbolics

3600 Software: Environment

The 3600 does not have an operating system in the conventional sense, but all of the functionality provided by an operating system is provided by some part of the software. This is because the software is designed as an integrated Lisp environment for a single user, rather than as a timesharing system that has to divide the computer between several users and protect each user from the others.

On the 3600, no rigid distinction exists between the operating system proper and fixed system utilities. System functions and utility functions are integrated and intermingled. For example, parts of the Zwei editing system underlie both the Zmacs editor and the Zmail facility. Unlike conventional operating systems, the 3600 environment encourages users to extend the behavior of most utilities with their own Lisp code.

Hence, the software provides a multiprocessing virtual operating system functionality with a structure designed to benefit the Lisp environment. This stands in contrast to building a conventional operating system and then trying to create a Lisp system on top of that.

This chapter describes the main components of the 3600 software environment.

The Window System: The User's Perspective

The bit-mapped display screen is managed by the window system. *Windows* are rectangular regions of the screen that can be fully visible, partially visible/partially covered, or wholly covered by other windows, like pieces of paper on a desk. Every interaction that you have with the 3600 occurs in a window. The screen always contains one or more windows.

You can use windows to control many activities at once. Each activity is represented by a different window. Switching between activities is accomplished by simply clicking on the activity's window with the mouse, selecting an item from a menu, or typing on the keyboard. When one window is partially covered by another, you can click on the part of the window that is visible, and that window will come to the top. No state information is lost when switching between windows.

Rapid context-switching is important in many applications. In developing programs, you can split the screen between an editor window in the top half of the screen and a Zetalisp interaction window in the bottom half. When you find problems with your code, choose the editor window and start editing the corrections. After editing, one simple Zmacs command incrementally compiles those parts of the program that have changed. Then return to the Zetalisp window and try the program again.

The configuration of windows is controlled with the Screen Editor, an interactive system that lets you create, kill, move, and reshape windows.

A Hierarchical Window System

The window system is hierarchical: in the same way that the screen can be divided up into windows, a window itself can be further subdivided into smaller windows. A window that is divided in this way is called a *frame*, and the subwindows are called *panes*.

The size of a frame can be changed. The panes of the frame change their size within the new dimensions. The individual panes within a frame can also be manipulated.

Menus and Choices

Users interact with the window system by selecting options from menus which appear on the screen. Menus are lists of mouse-sensitive choices, surrounded by a border. The system has several styles of menus, including the following common ones.

- Momentary menu
  Each item is a possible choice. Positioning the mouse cursor over an item and then clicking the appropriate button makes the choice. Once the choice has been made, the menu disappears. Moving the mouse cursor outside the menu border without having made a choice also makes the menu disappear. (See figure 13.)
- Choose-variable-values menu
  Each line presents a number of possible values of a particular parameter. The presently selected value appears in bold face. Each of the values is mouse-sensitive. Usually clicking on a

Figure 13. Example of a momentary menu.

Windows	*This window*	*Programs*
Create	Attributes	Lisp
Select	Refresh	Edit
Split Screen	Bury	Inspect
Layouts	Kill	Mail
Edit Screen	Reset	Font Edit
Set Mouse Screen	Arrest	Trace
Color Window	Un-Arrest	Emergency Break
		Hardcopy

Figure 14. Example of a choose-variable-values menu.

```
Edit window attributes of Lisp Listener 1.
Current font: CPTFONT
More processing enabled: Yes No
Reverse video: Yes No
Vertical spacing: 2
Deexposed typein action: Wait until exposed Notify user
Deexposed typeout action: Wait until exposed Notify user Let it happen Signal error Other
("Other" value of above): NIL
ALU function for drawing: Ones Zeroes Complement
ALU function for erasing: Ones Zeroes Complement
Screen manager priority: NIL
Save bits: Yes No
Label: Lisp Listener 1
Width of borders: 1
Width of border margins: 2
Done □                                    Abort □
```

different value selects it. These menus have explicit finishing commands, labeled Do It and Abort. You must select one of these in order to make the appropriate action occur and the menu disappear. (See figure 14.)

The Mouse Documentation Line

Near the bottom of the main screen is a one-line, inverse video window called the mouse documentation line. When the mouse is positioned over an item of a menu, the mouse documentation line indicates, for each of the three buttons, what would happen if that button were clicked. As the mouse is moved from one window to another, or as windows pop up or change configuration under the mouse, the mouse documentation line

changes to signal the new meaning of the mouse buttons in the new context.

Figure 15. The mouse documentation line and the status line.

```
Create new window.  Flavor of window selected from a menu.
01/12/83 03:31:11 ROADS        USER:        Tyi__
```

The Status Line

Below the mouse documentation line is the status line, which displays the following:

- The date and time
- The name of the user logged into the machine
- The state of the current process: whether it is running, paging, stopped, waiting for keyboard input, or waiting for any of various other things to happen
- The current package in which Lisp expressions typed in from the keyboard will be read
- Notification of file transfers in progress, with the name of the file and the percentage being read or written
- The keyboard idle time

Multiple Display Screens

More than one bit-mapped display screen can be attached to the 3600. The window system works equally well on all of them. All of the facilities provided by the window system, including menus, frames, and the Screen Editor, work on any display, including color displays. The mouse tracks on any screen.

The Window System: The Programmer's Perspective

Programmers use the window system to manipulate windows, implement graphics, perform input/output operations, and create a consistent graphical user interface to a program.

The window system is implemented with flavors (see page 38).

Each window is a Zetalisp object, an instance of some flavor. To manipulate a window, a program sends it messages. Messages perform the following kinds of tasks:

- Examine and alter the shape and position of a window
- Control the appearance of the borders and the label
- Manage stream and graphical input/output
- Control the cursor position
- Change the type font and control the interline spacing
- Insert and delete lines and characters

Primitive Graphics Operations

Windows also understand a large set of messages which perform primitive graphics operations (in color or black-and-white), such as drawing points, lines, filled-in rectangles, filled-in triangles, regular polygons, circles, curves described by a sequence of line segments, and cubic splines.

Another important graphics primitive provided is **bitblt** (known as RasterOp on some other systems), which moves arbitrary rectangular sections of a picture between arrays and windows or within windows, combining bits using a logical operator. For high speed, **bitblt**, rectangle, and triangle drawing are implemented with microcode support.

Choice Facilities

Using interactive graphics techniques, the 3600 provides a variety of choice facilities. By calling simple predefined functions, programmers can create windows that allow users to select items in a number of different ways. The following are some of the 3600's choice facilities.

- Momentary menus

  Momentary menus appear on the screen with a list of choices. The user does not have to make a choice, however. By moving the mouse outside of the menu, the user can make the menu disappear. Many options are provided.

- ↗ Pop-up menus
  Pop-up menus are like momentary menus except that they do not disappear until the user makes a choice. When the choice is made, the menu disappears and the chosen item is executed. The value of that object is returned. One type of pop-up menu is supplied with defaults and is easy to invoke. The second type can be customized.
- ↗ Command menus
  Command menus are used when you want to pass a command to a controlling process from a menu. The command is sent to the process via an input/output buffer which may be shared with other windows or processes. This way, the controlling process can be looking in the buffer for commands from several windows as well as for keyboard input. Optional features are available.
- ↗ Dynamic item list menus
  A dynamic item list menu is provided for menus whose items change over time. One message-passing operation updates the item list.
- ↗ Multiple menus
  Multiple menus are provided for situations in which the user can select *several* items at a time. These can be displayed in inverse video. Special choices allow the user to specify operations on all the selected items. Both momentary and pop-up menus are available.
- ↗ Multiple choice menus
  This facility displays a menu in which each item is associated with several yes/no choices, in choice boxes. This facility is supplied with many reasonable defaults.
- ↗ Choose variable values
  In this kind of menu, each item is associated with a value printed next to it. By selecting an item with the mouse, the user can alter the value of the item. Many defaults are supplied, yet facilities for customization are also available.
- ↗ User options
  The user option facility is based on the choose-variable-values facility. It is used to keep track of options to a program of the sort that users would want to specify once and then save. The option list can be associated with particular programs.

- Mouse-sensitive items

  This facility is similar to the choose-variable-values feature, but it is different in the way it is accessed by a program. This facility lets areas of the screen be sensitive to the mouse. Moving the mouse into such an area causes a box to be drawn around it. At this point, clicking the mouse invokes a user-defined operation. Many different kinds of mouse-sensitive items can be specified.
- Margin choices

  Windows can be augmented with choice boxes in their margins. Choice boxes give the user a few mouse-sensitive points which are independent of anything else in the window. Margin choices can be added to any flavor of window in a modular fashion.

Scrolling

Figure 16. The double-arrow symbol indicates that scrolling mode is in effect. The thickened line is a proportional representation showing the current screen's size and position in the entire buffer.

```
(DEFPROP :CHARACTER-OR-NIL
          (CHOOSE-VARIABLE-VALUES-CHA
           CHOOSE-VARIABLE-VALUES-CHA
           NIL NIL NIL "Click left to
          CHOOSE-VARIABLE-VALUES-KEYW
(DEFUN CHOOSE-VARIABLE-VALUES-CHARAC
  (FORMAT STREAM (IF VALUE "~:@C"
```

Many windows in the system respond to scrolling commands, invoked by moving the mouse cursor into one of the margins of the window (see figure 16). The software interface for scrolling is a flavor mixin (see page 42) and is available to use in any program. This makes it easy to create windows that handle scrolling.

Creating New Windows with Mixin Flavors

Many programmers construct the user interfaces to programs by using the Flavor System to combine their own flavors with others chosen from the window system's extensive library (see page 42). The private component flavors control or modify the behavior of predefined flavors to meet the needs of the specific program. For example, you can create flavors to control the way a window redisplays or how it responds to shaping. You can write and send messages that augment the predefined window-system messages.

File Systems

Symbolics computers use disks for file storage. Disks attached to 3600s contain the Lisp Machine File System (LMFS). Both local and remote file systems are supported on the 3600.

Figure 17. Kinds of file systems.

	Computer System	
	3600	Other Computer
Local File System (on 3600)	Local LMFS	—
Remote File System (other computer on the Ethernet)	Remote LMFS	Remote UNIX, VMS, TOPS-20, . . .

The Lisp Machine File System

The Lisp Machine File System (LMFS) can be used in a local file system, in a remote file system, or both. Features of the LMFS include:

- Conservative, robust design for maximum reliability
- Tree-structured directories
- File names of any length
- Arbitrary user properties attached to a file and used by user programs
- Version numbers

- Generation retention counts for automatic deletion
- Soft-deletion and expunging of files
  - Deletion just makes a file invisible
  - Undeletion can be used to recover it
  - Expunging eliminates a file entirely from the disk
- Noncontiguous disk-block allocation
- Large number of directories
- Multiple disk support
  - File system *partitions* (parts of a file system) can reside on different disks
- Reliable file salvager that can be run while the file system is in use
- Flexible backup facilities providing complete, incremental, and consolidated dumps
- Demountable disks for offline storage
- Links to other files in the file system

Up to four 474-Mbyte disks can be added to a 3600 with a single input/output board to create a large file system.

File System Reliability

The file system is characterized by a conservative, robust design. All file system information is redundantly stored so that loss of a single block of the disk cannot damage more than one file. Directories can be completely recreated from information stored in the file headers of the files. Every block of data in the file system is marked with a unique identifier, so that the file system can check to make sure that when it has read a block, it has obtained the right data.

The file system is written using a transaction discipline so that users can continue using it after a crash without running the Salvager. The Salvager is provided for long-term recovery of "lost" blocks, rebuilding damaged directories, and otherwise automatically fixing file system problems. It is simple to use and can be run even while the file system is operating.

A magnetic tape backup system is also provided, supporting complete, incremental, and consolidated dumps. Because of the software's flexibility, it is possible to route backup to any tape drive on the Ethernet, as long as it is attached to a

Symbolics-supported host operating system (for example, 3600s and computers running UNIX).

Remote File Systems

With the 3600's remote file system capabilities, users can access the file systems of other computers over the Ethernet. These remote file systems can be based on 3600s or timesharing computers. Many remote file systems can exist on a given network. Among the features of the 3600 remote file system software are the following:

- Generic file system access is provided across the Ethernet.
- Uniform pathname conventions are followed.
- Pathnames are implemented as flavor objects for flexibility.
- Directory editing is supported via the File System Editor and Dired.
- Wildcard matching and automatic file-name completion are provided for supported file systems.

Any command that accepts a file name, or any function that takes a file name as an argument, can accept a file name for any file system, in that system's own syntax. Thus, you can read files into editor buffers, or read them from any program, specifying the file name in whatever syntax the file server uses. Files from different servers, all with different naming conventions, can be accessed simultaneously. For example, a simple copy command can transfer a file between any two network hosts.

Users' programs can take advantage of the generic file system features. Files are instances of generic objects implemented using the Flavor System (see page 38). The result of opening a file is also an object, with all the properties of a stream (see page 35), plus other properties specific to the device it is on.

The File System Editor works on any supported file system, local or remote. File systems currently supported include those under VAX/VMS and UNIX (Berkeley Virtual).

Support for Logical File Names

In order to simplify the writing of portable software, the file system permits a distinction between a logical name for a file and the actual name of a file, given in the user-defined site file. Programmers can write all file input/output routines using the logical file name in their programs. The actual pathname of the file can be stipulated differently at a number of sites.

For example, in Symbolics system software, files are specified using logical host names in the following format:

```
host: directory; file-name type version
```

The actual pathname of the same file is specified in the user-defined site file. Thus, the actual name of a file on the 3600 would be specified in the following format:

```
host:>directory-path>file-name.type.version
```

The actual pathname of a file on a UNIX system would be specified differently:

```
host:/directory-path/file-name.type
```

Virtual Memory: The Programmer's Perspective

Virtual memory management is an integral part of the 3600's design. Users are provided with a very large (256-Mword or 1-Gbyte) virtual memory address space. It is not organized on a per-process basis. Rather, the virtual memory is split up into areas by the area feature of Zetalisp, and further divided into Lisp objects. System software in compiled form occupies approximately 4 Mwords of virtual memory. The rest of the address space is available for system tables and user programs.

Virtual memory is managed by a paging algorithm that approximates the least-recently used (LRU) page-replacement principle, using the standard clock algorithm. Normally, users do not need to be concerned with its operation. The system does, however, provide certain options.

- Paging policy can be either the normal area access or sequential access. Sequential access provides a "double-buffering" capability, since the paging system automatically prefetches sequentially accessed pages.
- For high-speed operation, critical pages can be wired

(resident) in main memory, thereby avoiding the replacement algorithm.[4]

- The swap-in quantum (number of pages to prefetch) is adjustable on a per-area basis.

Hardware, microcode, and system software share in the task of virtual memory management of the 3600. (For more detail, see page 94.)

Scheduling Processes

The 3600 features an efficient scheduler which provides a number of services:

- Real-time processes, wired in main memory
- Fast response for interactive activities
- Background activities
- Priority setting under user control

A number of processes can run concurrently. The scheduler implements processes with the stack group coroutining mechanism of the Zetalisp language. A process can wait for any arbitrary condition to occur. To wait for a condition, a process passes the scheduler a waiting function. The scheduler periodically calls this function, and as soon as the function returns a true value, the process is allowed to proceed. This is the fundamental blocking mechanism; some higher level facilities, such as sleeping (for a given amount of real time) and locks (binary semaphores), are also provided.

Programmers can also construct more complex multiprocessing control structures. It is easy for processes to communicate data to one another, since they all exist in the same Lisp world. Programmers can build any kind of mailbox, queue, or other multiprocessing facility. Some simple ones are supplied by the system.

The Lisp software environment uses processes heavily.

- One process reads keystrokes from the keyboard, handling interrupt characters and directing input to appropriate windows.

---

[4] In other systems this is called "locking" or "fixing" pages.

- Two processes control the reception and transmission of packets on the network.
- The network remote login programs (Telnet and Supdup) work by splitting into two processes: one to transmit to the foreign host, and one to receive from the foreign host.

Users rely on processes as well. When they create a new window to run an interactive system, such as a Lisp Listener window or a Zmacs window, a new process is created to run that program.

Lisp programs deal with processes by calling a function to create a process. The programs then send messages to the process to start, stop, examine, and alter the state of the process. A simple Lisp function is provided (**process-run-function**) that calls another function on some arguments in a newly created process. Any time a program requires multiprocessing, it can easily spawn any number of new processes to run concurrent programs. The scheduler is genuinely preemptive; it is co-called periodically by a hardware-generated interrupt. The time-slice period is under user control.

Support for Real-time Processes

Users of the 3600 can create efficient real-time processes to control external devices. This is because on the 3600 a real-time process is a special object. The scheduler treats real-time processes differently so that such processes can respond quickly to external stimuli, such as real-time clocks or device interrupts.

As one of its special features, a real-time process never takes page faults. All of its code and data structures are wired in main memory. In addition, the 3600 processor maintains a special hardware stack buffer, so that real-time processes can run without having to save away the stack buffer of the main process. Hence, a real-time process can be started quickly because the hardware does not have to perform a full process switch.

The 3600 provides other kinds of support for devices that require fast response. For example, microcode tools are provided for the support of devices on the L bus. Interrupt handlers for

MULTIBUS devices can be written in the LIL language and run on the FEP.

3600 Software: Network Communications

This chapter surveys Zmail, Converse, the Symbolics Auto-dial Feature, and other network products.

Electronic Mail: Zmail and Converse

In a modern computing environment, powerful electronic mail facilities are a necessity. On timesharing systems, intercommunication is limited to users of the same machine unless that machine is on a network with other computers. The new generation of electronic mail systems allow users on different network-linked computer systems to communicate with the same ease as users on a single machine.

Zmail is an elaborate interactive network-based mail facility. Zmail's command interface is easy to learn. You can start by learning a basic set of commands, accessible through a user interface based on graphical windows and menus. Zmail provides many more advanced and powerful commands to help you keep track of and handle large amounts of mail.

New mail from other users is shipped to a user's inbox. When recipients read their mail, the contents of their inbox are appended to their default mail file. Each message in the default mail file can either be deleted, replied to, or stored in a mail file. (See figure 18.) Messages can be automatically filtered on the basis of certain keywords in the message, on the contents of the message header, or on the basis of certain properties of the message (for example, whether it has been answered).

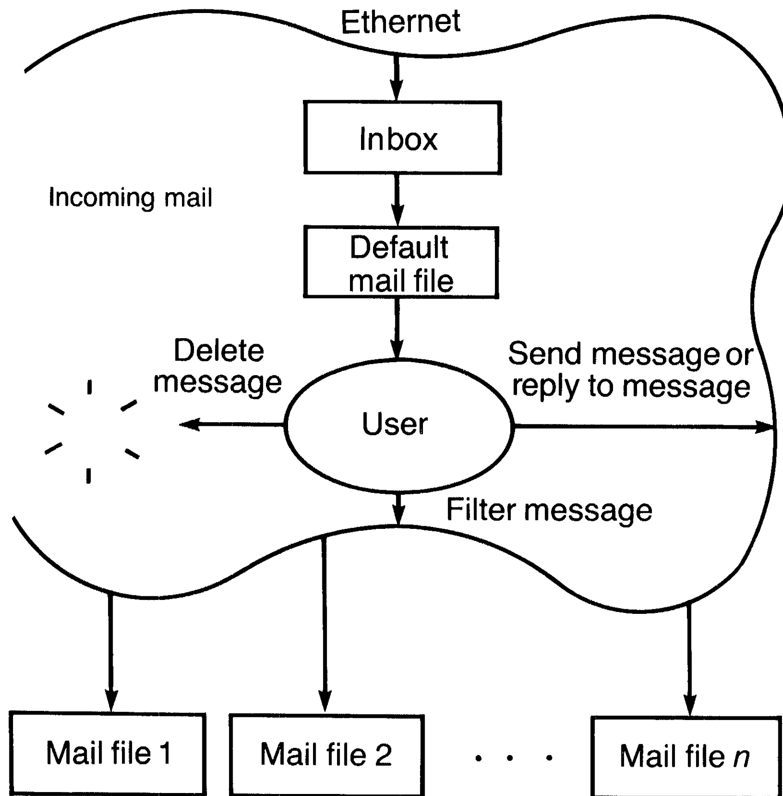
Reading and Answering Mail

When you peruse through mail with Zmail, the top of the Zmail window contains a summary of the messages in their mail file (see figure 19). A summary line of descriptive text is shown for each message. One of these messages is the currently selected message, and its summary line is marked with an arrow. In the bottom of the Zmail window is the actual message, in its entirety if it fits. Scrolling is provided for long messages.

Between the summary and the message is a menu of commands which perform the following operations.

↗ Get next message

Figure 18. Zmail input and output diagram.



- Get previous message
- Delete a message
- Undelete a message
- Reply to a message
- Jump to a message
- Move a message to a buffer
- Change the keywords associated with a message
- Survey messages
- Sort messages
- Get incoming mail buffer
- Send new mail
- Invoke a menu of operations to perform on a group of messages: Delete, Type out, Move, Redistribute, Concatenate, Find strings, Forward, Reply to, Put keywords on or take them off
- Execute an auxiliary command: Whois or ViewFile

- Edit user's Zmail profile
- Change the window configuration of Zmail
- Save or expunge mail buffers

You can move through messages by using the Next and Previous menu commands, by pointing the mouse cursor at a line in the summary window and clicking the mouse, or by typing single-character commands on the keyboard.

When replying to mail, you can see the message you are replying to in the top half of the window. The reply goes in the bottom half.

Full Zmacs editing capability is provided in Zmail. In the Zmail environment, the editor is extended with special mail-oriented commands.

Selecting and Filtering Mail

Several mail files can be read at a time, and you can move messages from one mail file to another. This is so mail about different topics can be saved in different mail files. You can also select sets of messages on which some operation should be performed. For example, you can select a set and then delete all the messages in the set, or move them all into a mail file.

The set can be specified either by pointing at its lines in the summary window or by using *filters*. A filter is a set of criteria used to determine whether a message should or should not be accepted into a set. You can filter messages based on such attributes as the sender, the recipients, the time of sending, the contents, or whether they have replied to the message. You can also combine these criteria using AND, OR, and NOT predicates to specify complex filters that discriminate between messages. User-written Lisp programs can also be used as filters.

Customizing Zmail

Zmail is readily customizable. You can control the layout of the display, the format of paper copies of mail files, and many other options. As discussed, you can define special-purpose filters to access specific categories of messages. The contents of Zmail menus and the defaults for commands with options can be

Figure 19. Zmail window in a mail-reading state.

```
 No. Lines  Date  From→To                        Subject or Text
  46:   24 15-Nov MMcM→Margolin@MIT-MULTI        Setting the From: field
  47:   26 17-Nov RWK→BUG-ZMAIL                  ZMAIL  9
  48:   25 19-Nov Steve Pelaggi→scrc             Time cards
  49:   29 22-Nov dlw→                           Exception handling
  50:   48 22-Nov Min→Symbolics                  Foreign Distributorship
  51:   25  1-Dec DCP@SCH-HUEY-DU→network        As most of you know by now, the land lines (SCRC<-->S
  52:   67  4-Dec Jwalker→Whit                   protocol for camera-ready copy
  53:   16  5-Dec →whit                          Zmail manual; NES manual
  54:   12  7-Dec finkel→Lang                    Society for Technical Communication
  55:   27  8-Dec finkel→JWalker                 Toolkit
  56:   22 13-Dec whit→scrc                      Release 4.0 Beta-test Site Documentation review files
  57:  116 13-Dec whit→info-lispm,doc            Documentation and Training Services project summaries
  58:    8 15-Dec cec→scrc                       Customer file
  59:   59 19-Dec RWK→bug-zmail                  What is incremental expunging?
  60:   19 20-Dec JAYNE→scrc,jean                Prudential Open Enrollment
  61:   11 20-Dec nuwav→scrc                     Organization
  62:   17 20-Dec pmiller→symbolics              CEC maintains CUSTOMERS.TEXT
  63:    9 20-Dec nuwav→scrc                     Conference
  64:   28 28-Dec RWK→                           Disappearing fonts
  65:   13 29-Dec BSG→rn                         Object-oriented programming
  66:   26 30-Dec jwalker→dlw                    documentation copies
▸ 67:   15 30-Dec linda→sch,scrc,spa,stx.        "An Advanced Lisp-Based Engineering Workstation"

   Profile           Quit          Delete      Undelete      Reply
   Configure         Save          Next  X     Previous      Continue
   Survey            Get inbox     Jump        Keywords      Mail
   Sort              Map over      Move        Select        Other

Date: 30 Dec 1982 15:20:17-EST
From: linda at scrc-vixen
To: sch at scrc-vixen,  scrc at scrc-vixen,  spa at scrc-vixen,  stx. at scrc-vixen
Subject: "An Advanced Lisp-Based Engineering Workstation"
Cc: linda at scrc-vixen


Copies of the above paper written by J.L. Kulp and David Schwartz
are available either through me or in the documentation files in the
Library.  Please comment if so desired.

Regards,

linda










Message
Zmail SCRC:<JLH>JLH.BABYL     Msg #67/67 (last)  ()



01/05/83 14:57:10 JLH          USER:      Tyi___
```

customized. Zmail makes it easy to customize these things without having to understand how they are implemented, by providing a display-based user-profile editor.

The Converse Utility for Interactive Messages

Interactive messages can be sent to other users on the network using the Converse utility. When the system receives a message, it pops up a window, displays the message in the window, and prompts the recipient for a response. That user can then type a message and it will be sent off to the original sender.

Distinguishing features of Converse include:

- Full editing capability when typing messages, including extended commands
- Messages can be routed for wider distribution through Zmail as well as to the immediate addressee
- Multiple, simultaneous conversations

Network Software

Extensive network software is available with the 3600, including support for Ethernet local area networks and remote networks.

Ethernet Support

The 3600 offers complete support of the 10-Mbit/sec Ethernet, a low-cost, highly reliable local area network. Features include:

- Packet-switching
- Carrier-sense, multiple access/collision detection (CSMA/CD)
- Baseband transmission
- Coaxial cable connecting all network nodes

A main advantage of the Ethernet is the ease with which new nodes can be attached to the net. The Ethernet transceiver is hooked to a computer and attached to the coaxial network cable. 3600s, as well as other computers, can share the network.

Control of the Ethernet is distributed among the communicating computers to eliminate the reliability problems of an active central controller. The costs of a central controller are also eliminated, making smaller networks more feasible.

The Symbolics Network System provides protocols for Ethernet

hardware. Lisp programs have full access to the facilities of the network. Lisp functions are provided to open network connections as a user or to handle incoming network connections as a server. User programs can accept, reject, and close connections, and transmit or receive packets. A network connection can also be handled as an input/output stream, with standard stream input and output message-passing operations. Programmers can invent higher level network protocols or use existing ones. The network control program provides error-checking, flow-control, and retransmission, and maintains the multiplexing and demultiplexing of the network into many separate connections.

The Symbolics Network System also has provisions for the following:

- Simple transactions: a single exchange of packets, with acknowledgement, without forming a connection
- Uncontrolled packets, without flow control and acknowledgement
- Broadcasting
- Automatic redistribution of routing information for gateways
- Error reporting

Higher level protocols exist for the following functions:

- Remote login
- File access
- Mail transmission
- Interactive messages
- Finding out the users of a machine
- Finding out the current time
- Accessing remote printers and tapes
- Opening a gateway to remote networks
- Network maintenance and testing

Network control programs also exist for VAX/VMS and UNIX (Berkeley Virtual).

Local network cables can be extended up to 500 meters, with up to 100 transceivers per cable.

Remote login

Standard software includes support for remote login over large-scale networks. For example, users can log in to other computers over a large-scale network from a 3600 console. The 3600 supports the standard Telnet remote-login protocol, as well as the Supdup display-terminal protocol.

Symbolics Auto-dial Feature

The Symbolics Auto-dial Feature permits the direct connection of 3600 computers over standard telephone lines. This also allows two 3600s that are part of different local area networks to connect and act as a gateway between sites.

The Auto-dial Feature places Symbolics in direct digital connection with all of its customers. This facilitates:

- Customer support
- Remote diagnosis of 3600 systems
- Exchange of such information as bug reports from customers and patches from Symbolics
- Software and documentation distribution

High-level services previously available only to computers linked on a local area network are provided between computers on different networks. Services include:

- Remote file-system access
- Remote peripheral access to magnetic tape, disk, and printer
- Electronic mail and real-time interactive message sending
- Remote host system status and user name-server facilities
- Remote login with virtual terminal support

YES ↑ NO ↓ ENTER RUN FAULT

SECURE LOCAL REMOTE SECURE

OFF REMOTE

RESET POWER

symbolics 3600

3600 Hardware: A New Lisp Machine

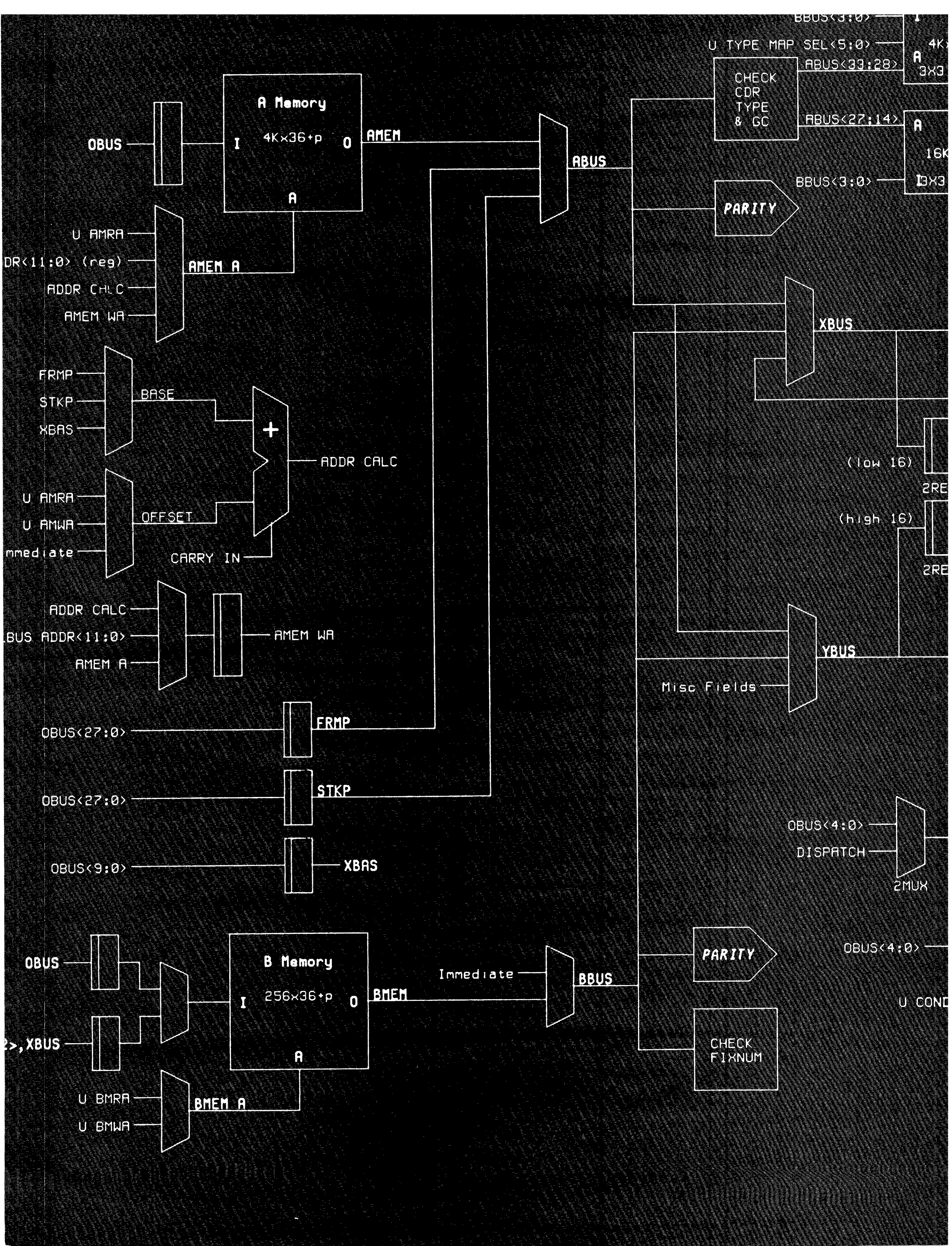
The 3600 is built to execute large programs which need high-speed symbolic and numeric computation. Because the 3600 hardware and software were designed in parallel, the instruction set of the machine is designed for efficient execution of Lisp. Many Lisp instructions execute in one or two microcycles.

The 3600 is not simply a faster version of the older Lisp Machines. The 3600 features an entirely new hardware design which takes advantage of many technological advances of recent years. For example, the 3600 is the first computer to provide hardware-assisted garbage collection.

High-speed computations in the 3600 are aided by the use of a number of caches designed to increase Lisp performance, including the following:

- Instruction cache
  Used for prefetch of instructions
- Stack buffers
  Used for Stack manipulations
- Map cache
  Used for virtual memory operation
- Garbage collection tables
  Used for recovery of memory space

The result of this new design is a processor that is especially fast for Lisp but is also reliable. Reliability is accomplished through a myriad of automatic checks which do not slow down the execution speed of user programs.



BBUS<3:0>
U TYPE MAP SEL<5:0>
ABUS<33:28>
CHECK
CDR
TYPE
& GC
ABUS<27:14>
A Memory
4Kx36+p
OBUS
I
O
A
AMEM
ABUS
BBUS<3:0>
PARITY
U AMRA
DR<11:0> (reg)
ADDR CALC
AMEM WA
AMEM A
XBUS
FRMP
STKP
XBAS
BASE
+
ADDR CALC
U AMRA
U AMWA
mmediate
OFFSET
CARRY IN
(low 16)
(high 16)
ADDR CALC
LBUS ADDR<11:0>
AMEM A
AMEM WA
YBUS
Misc Fields
OBUS<27:0>
FRMP
OBUS<27:0>
STKP
OBUS<9:0>
XBAS
OBUS<4:0>
DISPATCH
2MUX
B Memory
256x36+p
OBUS
Immediate
BBUS
PARITY
OBUS<4:0>
BMEM
XBUS
CHECK
FIXNUM
U BMRA
U BMWA
BMEM A

3600 Hardware: Processor Architecture

Although the 3600 is a single-user machine, its processor is not in the microcomputer class of, for example, the MC68000. Rather, it is more of a "supermini" computer, with a raw computational power of over one million 32/36-bit operations per second in parallel with console display and FEP operations.

The 3600's processor architecture is significantly different from that of conventional systems. The features of the 3600 processor architecture include the following:

- Microprogrammed 32/36-bit processor designed for Lisp (180- to 250-nanosecond cycle time, variable)
- Run-time data-type checking in hardware
- Stack-oriented architecture
- Large, high-speed stack buffer with hardware stack pointers
- Fast (5 MIPS) instruction fetch unit with large instruction cache
- Efficient hardware-assisted garbage collection
- Microtasking
- 5-Mwords/sec peak data transfer rate (20-Mbytes/sec)
- 256-Mword (1-Gbyte) virtual memory

Parallelism is exploited in the 3600 processor by executing these operations concurrently:

- Run-time data-type checking
- Garbage-collection support
- Result tagging
- Instruction fetch
- Instruction decode
- Instruction execution

Tagged Architecture

The 3600 processor exemplifies a tagged architecture computer. In recent years, it has become recognized that some form of data-type checking is important to catch invalid operations before they occur. This ensures program reliability and data integrity.

Run-time Data-Type Checking

The special hardware allows data-type checks to be carried out at run-time and not just at compile-time. Data-type checking was performed in microcode in previous Lisp Machines. Hardware implementation of this operation on the 3600 speeds it up

considerably. This increase in speed is accomplished by performing the data-type checking in parallel with instruction execution, rather than requiring execution of extra microinstructions.

Run-time data-type checking is important in a dynamic Lisp environment, since compile-time data-type checking is not compatible with the flexibility of the Lisp language and the generic nature of most functions which allows them to operate on many different data-types. Garbage-collection algorithms (see page 98) are also aided by fast data-type checking.

Run-time data-type checking is supported by appending a *tag field* to every word processed by the processor. The tag field indicates the type of the object being processed. For example, by examining the tag field, the processor can determine whether a word is an integer or a floating-point number.

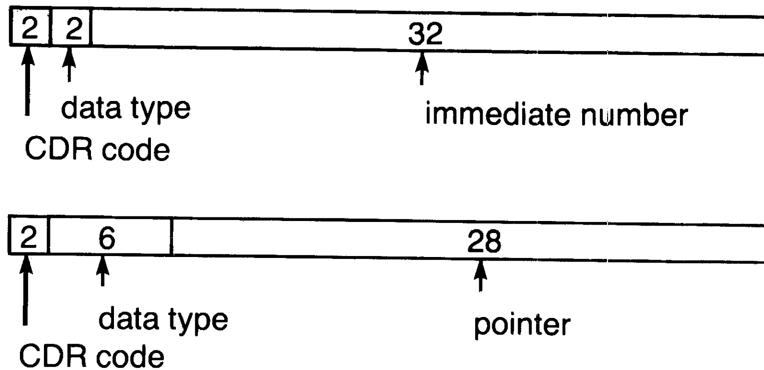
With the tagged architecture of the 3600, all macroinstructions are *generic*. That is, they work on all data types appropriate to them. For example, one add operation, is good for fixed- and floating-point numbers, double-precision numbers, and so on. The behavior of a specific ADD instruction is determined by the types of the operands, which the hardware reads in the operands' tag fields. No performance penalty is associated with the data-type checking, since it is performed in parallel with the instruction. By using generic instructions and tag fields, one 3600 macroinstruction can do the work of several instructions on more conventional machines. This permits very compact storage of compiled programs and fast execution.

Word Formats and the Cdr-coding Feature

On the 3600, a word contains one of many different types of objects. Two basic formats of 36-bit words are provided, as shown in figure 20.

One format, called the tagged pointer format, consists of a 6-bit tag, 2-bit cdr-code, and 28 bits of address. The other format, called the immediate number format, consists of a 2-bit tag, a

Figure 20. 3600 word formats.



2-bit cdr-code, and 32 bits of immediate numerical data.[5]

Cdr-coding for List Compaction

An important feature of the 3600 is the use of list compaction or cdr-coding. Understanding of this feature requires knowledge of the special Lisp terms for parts of list structure. The first element of a list is called the car; the rest of the list is called the cdr. The way lists are represented in traditional Lisp implementations is shown in figure 21.

On the 3600, the amount of storage needed to hold such a list can be reduced using a cdr-coded representation. Two bits of every 3600 word are reserved for cdr-coding.

The possible values of the cdr-code are normal, next, and nil. Normal indicates a standard car-cdr list element pair. Next and nil represent the list as a vector in memory. This takes up only half as much storage as the normal case, since only the cars are stored. Zetalisp primitives that create lists make these compressed cdr-coded lists. Figure 22 presents an example of the same list shown in figure 21, but with a cdr-coded representation. Notice that the number of memory cells needed to store the list is reduced from six words to three.

---

[5]In main memory, each word is supplemented with eight more bits, including seven bits for ECC (error correction code) and one spare bit, to make a 44-bit word.

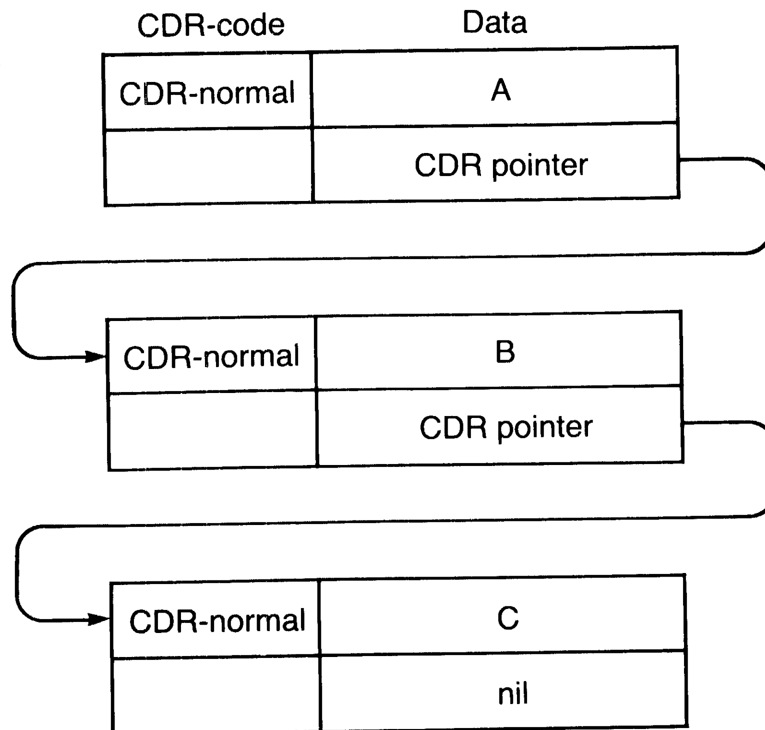


Figure 21. Representation of the list (A B C) in normal form uses six words. (Note that the type field is not shown in this diagram.)

Hardware-supported Data Types

Thirty-four data types are directly supported in hardware by the 3600 processor. The type-encoding scheme is shown in figure 20. A Lisp pointer is represented in 34 bits of the 36-bit word. The other two bits of the word are reserved for cdr-coding. The first two bits of the 34-bit tagged pointer are the primary data-typing field. The value of this field indicates whether the other 32 bits hold an immediate fixed-point or floating-point number. (The floating-point representation is compatible with the IEEE standard.) The other two values of the 2-bit field indicate that the next four bits are further data-type bits. The remaining 28 bits are used as an address to that object.

Figure 22. Representation of the list (A B C) in compacted (cdr-coded) form uses three words. (Note that the type field is not shown in this diagram.)

CDR-next	A
CDR-next	B
CDR-nil	C

The object types include:
- Symbols
- Lists ("cons cells")
- Strings
- Arrays
- Flavor instances
- Bignums (infinite-precision integers)
- Extended floating-point numbers
- Complex numbers
- Rational numbers
- Coroutines
- Compiled code
- Closures
- Lexical closures
- Nil
- Internal codes not seen by the user

Stack Mechanisms

The 3600 is not a pure stack machine, but it is a stack-oriented machine. This means that most, but not all, of the 3600 instructions use the stack in getting operands and storing the results. Multiple stacks and multiple stack buffers in hardware

are a part of the design of the 3600 processor. Active stacks are always kept in a stack buffer. The stack buffers in the processor provide fast temporary storage for data references associated with programs, such as values being computed, arguments, local variables, and control-flow information. A main use of a stack is to pass arguments to functions and flavor methods. Fast function calling is critical to the performance of processor-bound programs.

The 3600 features a new stack layout, designed to make function calls and returns as fast as possible. Maximum performance is achieved in the simple case of a function call with zero to four required or optional arguments.

A stack on the 3600 is managed by the processor hardware, which maintains various pointers to the stack. Stack-buffer manipulations such as push and pop are carried out by the processor and occur in one machine cycle.

Hardware Support for Stack Groups

On the 3600, a given computation is always associated with a particular stack group.[6] Hence, the stacks are organized into stack groups. A stack group has three components:

- Control stack
  Contains the control environment, local environment, and caller list
- Binding stack
  Contains special variables and lambda bindings
- Data stack
  Contains Lisp objects of dynamic extent, such as temporary arrays and lists

The control stack is formatted into frames. The frames correspond to function calls. A frame consists of a fixed header, followed by a number of argument and local variable slots, followed by a temporary stack area. The data stack is provided to reduce garbage-collection overhead.

---

[6]See David Moon and Daniel Weinreb, *Lisp Machine Manual* (Symbolics, Inc., 1981) for more details on stack groups.

3600 Instruction Set

The 3600 machine instruction set corresponds very closely to Zetalisp. Although programmers never program directly in the instruction set (no assembler is supplied with the 3600), they encounter the instruction set when using the Inspector or the disassembler.

Many 3600 instructions address a source of data on which they operate. If they need more than one argument, the other arguments come from the stack. Examples include PUSH (push source onto the stack), ADD (add source and the top of stack), and CAR (take the car of the source and push it onto the stack). These instructions exist in several formats.

Unlike LM-2 instructions, 3600 instructions do not have a separate destination field. All instructions have a version which pushes onto the stack. Additional opcodes are used to specify other destinations.

Instruction Formats

The instructions are 17 bits long, with nine bits for the opcode and eight for operand/address. They are packed two per word in memory. Seven instruction formats are used:

- Unsigned-immediate operand
  Used in immediate fixnum arithmetic and specialized instructions, such as adjusting the height of the stack. Operand is an 8-bit positive integer.
- Signed-immediate operand
  Operands in a similar manner as unsigned-immediate operand. Operand is an 8-bit two's complement integer.
- PC-relative operand
  Is similar to signed-immediate in branching and format. Operand is offset relative to the program counter.
- No operand
  Used by many basic Lisp instructions. Any operands not specified explicitly by the instruction are popped off the stack.
- Link operand
  Used for constants. Operand specifies a reference to a linkage area in the compiled-code object.

Table 1. Instruction Categories. (Note: The instructions listed here do not constitute the entire instruction set.)

Category	*Explanation*	*Examples*
Data movement instructions	Move data without changing them	**push-immed** **pop-n-save** **movem-local**
Instance variable instructions	Used in manipulating instance variables of flavors	**push-instance-variable** **movem-instance-variable** **instance-ref**
Function calling instructions	Call function; the arguments are already on the stack	**call-0-stack** **call-n-return** **funcall-1-stack**
Binding and function entry instructions	Used within functions that take more than four arguments or have a *rest* argument, and hence do not have their arguments set up by microcode	**take-n-args** **take-n-optional-args-rest**
Function return instructions	Return values from a function	**return-stack** **return-multiple**
Quick function call and return instructions	Call function quickly	**popj**
Branch instructions	Change the flow of program control	**branch** **branch-true-else-pop**
Catch instructions	Change the flow of program control with a nonlocal exit	**catch-open-stack** **unwind-protect-open**

Category	*Explanation*	*Examples*
Predicates	Standard tests	**eq** **not** **fixp** **loatp** **symbolp** **arrayp**
Arithmetic instructions	Standard arithmetic, logical, and bit-manipulation operations	**add-stack** **multiply-stack** **multiply-stack** **quotient-stack** **remainder-stack** **rot-stack**
List and symbol instructions	Used for manipulation of lists and symbols	**car** **cdr** **rplaca** **set** **symeval** **property-cell-location** **package-cell-location**
Array instructions	Define and manipulate arrays or access structure fields	**array-leader** **store-array-leader**
Subprimitive instructions	Provide access to memory, function calling, and various hardware operations at a level below the normal Lisp languages	**halt** **%multiply-double** **%data-type** **%pointer** **%stack-group-switch** **%gc-tag-read**

- Indirect link operand
  Used for function and special variable references. Operand specifies an indirect reference to the linkage area in the compiled-code object.
- Local operand
  Used for many basic Lisp instructions. Instruction addresses an argument in the local stack frame. Operands after the first are popped off the stack.

3600 instructions can be grouped into a number of categories as shown in table 1.

The Instruction-Execution Engine

The 3600 instruction-execution engine works as a combination of hardware and microcode. The engine includes hardware for the following functions:

- Address computation
- Data-type checking
- Rotation, masking, and merging of bit fields
- Arithmetic and logical functions
- Multiplication
- Stack-buffer manipulation
- Result-type insertion

Example of Instruction Execution: ADD

An ADD instruction causes the hardware to perform the following operations in one microcycle:

- Fetch the operands from the stack.
- Check the data-type fields.
- Assume the operands are integers and perform the 32-bit add; if the operands are not fixnums, trap to microcode to perform a different type of add.
- Check for overflow upon which trap to microcode to return a bignum result.
- Tag the result with the proper data type.
- Push the result onto the stack.
- Dispatch to the microcode for the next instruction.

Data-type checking incurs no overhead because it happens in parallel with the addition, within the same microcycle.

The Instruction Fetch Unit

A main goal of the 3600 architecture is to execute one simple macroinstruction per clock cycle. The instruction fetch unit (IFU) supports this goal by attempting to prefetch instructions and perform microinstruction dispatching in parallel with the execution of previous instructions.

The prefetch (PF) part of the IFU fills from memory a 1-Kword instruction cache. The instruction cache holds the 36-bit instruction words. 2K instructions (17 bits each) can be held in the instruction cache. The IFU takes the instructions, decodes them, and produces a microcode address. It also predicts branches and presents the instruction at the target of the branch, if it is in the cache, as the next instruction. Figure 23 presents a diagram of this process.

Microcode and Microtasks

The microcode for the 3600 is contained in an 8-Kword microcode memory. Each 112-bit microcode instruction specifies two 32-bit data sources from a variety of internal scratchpad registers. Users would normally not write microprograms, since many Zetalisp instructions are executed in one microcycle. Thus, compiled Zetalisp code often runs almost as fast as customized microcode would. However, support for user microcoding is provided.

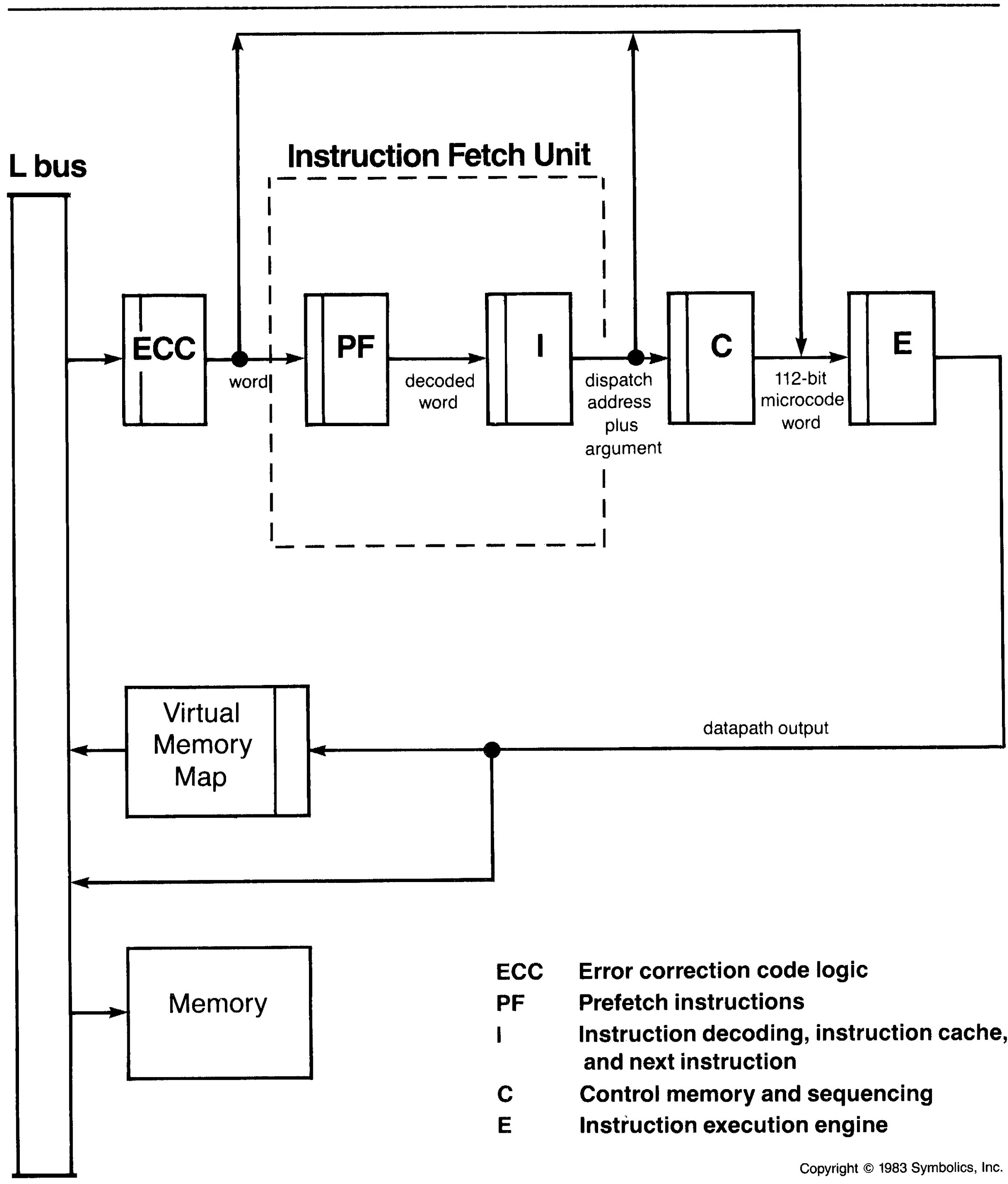
Microtasking

The 3600 micromachine is time-division multiplexed to support microtasking. This means that the processor performs input/output operations, such as driving the disk, in addition to executing macroinstructions. This has the advantage of providing the disk controller and other microtasks with the full processing capability and temporary storage of the 3600 micromachine.

Up to 16 different hardware tasks can be activated: eight for direct memory access (DMA) devices, two for other hardware, five for software, and one unused. Control of the micromachine typically switches from one task to another every few microseconds. The following microtasks run in the 3600:

- Zetalisp emulator

Figure 23. Instruction pipeline path, showing the relation between the IFU and the rest of the processor.



Copyright © 1983 Symbolics, Inc.

Executes instructions
- ↗ Disk transfer
  Manages disk operations
- ↗ Ethernet
  Performs network operations
- ↗ Audio output
  Maintains audio buffers and generate sound samples
- ↗ FEP
  Communicates with the FEP and generates interrupts at regular intervals
- ↗ Device service
  Controls DMA tasks

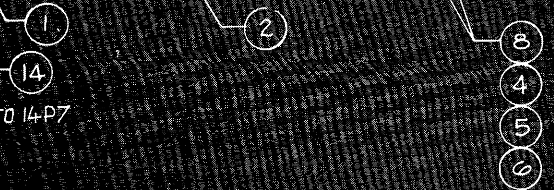
Microdevices are those devices serviced by microcode, such as the disk controller and the Ethernet controller. The FEP and the microdevices can initiate task switches on their own behalf, without incurring overhead. Logic on the sequencer board determines the priority of the microtasks. Multiple microcontexts are supported in hardware, eliminating the need for software to save the context of one microtask before switching to another.

symbolics inc.

PCBA 170002 (512K) OR 170003 (256K) SERIAL REV

JVB 82

P1 P2 P3



CH: R2 TO 21R6; 35L17 TO 34L15, AT 15R15
NS ON COMPONENT: 20P8 & 19R9
IRE: 21R6 TO R2 (AS SHOWN); COMPONENT PIN 20P8 TO 20P10
COMPONENT PIN 19R9 TO AA25; 19E5 TO 20E5; 19E6 TO 20E6;
19E7 TO 20E7, 14R11 TO 16R3, 15R13 TO 14R14, 14R15 TO 16R12, 35L17 TO 34L14, 14R1 TO 14P7
LIED WITH 128045
4: DESIGNATIONS C2 THRU C165 OMITTED FOR CLARITY.
: PART NO. NOT SHOWN ON FACE OF DWG. (SEE LOCATION BLOCK IN P/L)

3600 Hardware: Organization of Memory

The 3600 processor has a collection of interacting internal storage features for increasing computational throughput. These include the following:

- Scratchpad memory
- High-speed instruction cache
- Two large stack buffers
- Virtual memory addressing

This chapter describes these memory systems and the operation of the 3600's L bus.

Instruction Cache

The 3600 has a 1-Kword instruction cache which stores 2K instructions. The instruction cache is fed by the IFU (see page 89). It is cycled at the machine clock rate.

Stack Buffers

Because it is a stack-oriented machine, the 3600 has no general-purpose registers in the conventional sense at the macroinstruction level. This means that many instructions fetch their operands directly from the stack.

The two 1-Kword stack buffers are a special 3600 hardware feature which speed instruction execution. The stack buffers function as special high-speed caches used to contain the top portion of the Lisp control stack. Since most memory references in Lisp programs go to the stack, the stack buffers provide very fast access to the referenced objects. The virtual memory system will automatically map active processes into the stack buffers.

The stack buffers store several pages surrounding the current stack pointer, since they most likely contain the next-referenced data objects. When a stack overflows or underflows the stack buffer, a fresh page of the stack buffer is automatically allocated, possibly sending another page out to main memory.

Hardware Pointers

Another feature of the stack buffers, hardware-controlled pushdown-pointers, supports high-speed access to the stack. These pointers eliminate the need to execute microcode instructions to manipulate the stack. All stack manipulations

work in one cycle. A hardware top-of-stack register facilitates quick access to that location at all times. (See page 83.)

Physical Memory

Physical memory on the 3600 is addressed in 44-bit word units. This includes 36 bits for data, seven bits for error correction code (ECC) plus one bit spare. Double-bit errors are detected, while single-bit errors are both detected and corrected automatically. 3600 memory is implemented using 200-ns 64-Kbit dynamic RAM (random access memory) chips. The minimum memory configuration is 256 Kwords (1 Mbyte). The maximum physical memory configuration is 7.5 Mwords (30 Mbytes).

For random access to virtual addresses, the 3600 can read or write a word every three cycles (600 ns). Sequential addresses can be read or written every cycle (one word per 200 ns).

Virtual Memory

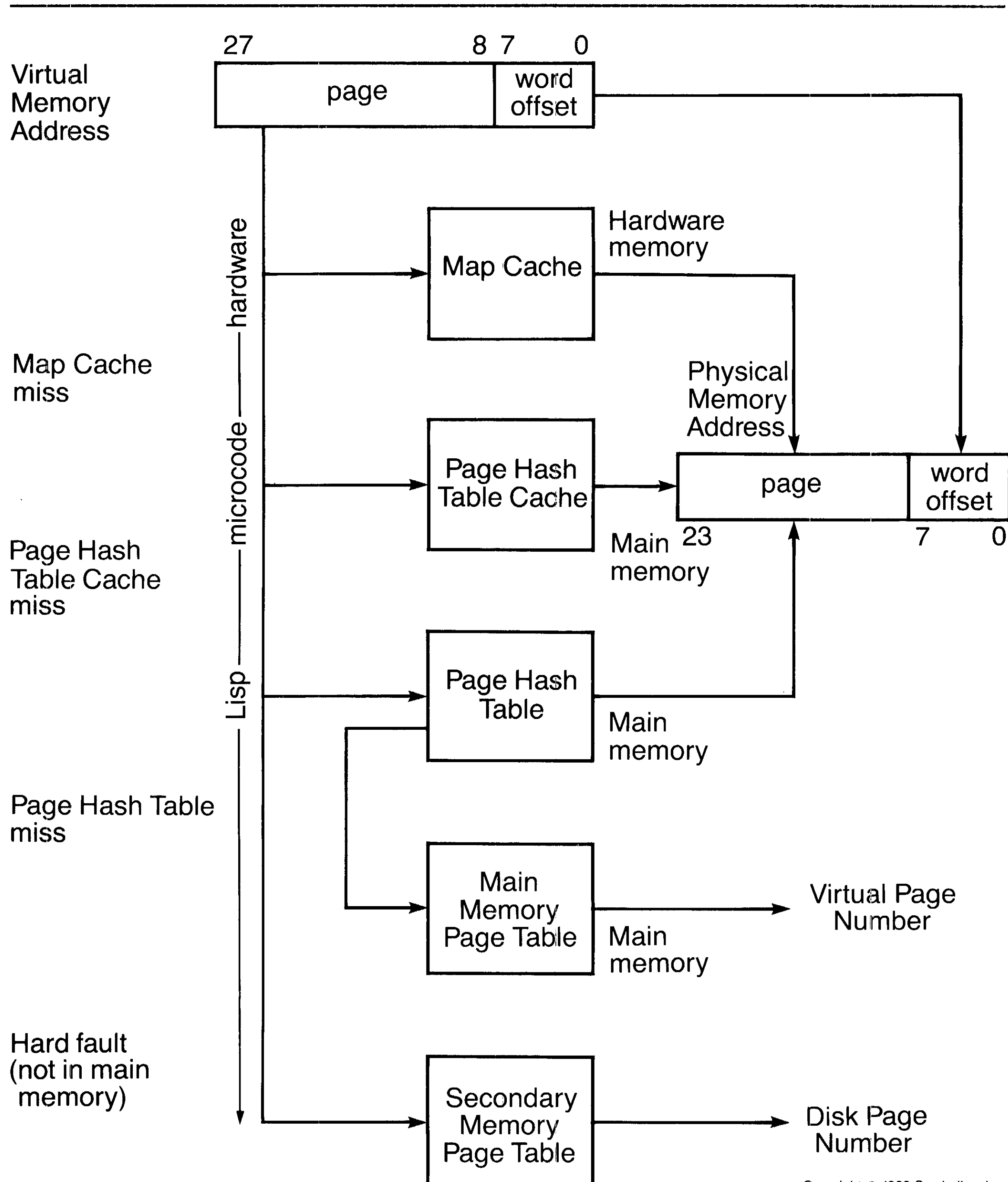
From the programmer's viewpoint, virtual memory is the space in which programs and data are contained (see page 65). From the system's viewpoint, the virtual memory space is a collection of pages of information, some of which reside in physical memory and some of which reside on disk. Since programs can be run only when they are resident in physical memory, the 3600 hardware automatically swaps between physical memory and the pages on the paging disk.

The 3600's 28-bit virtual address space consists of 256 million 36-bit words. The virtual address space is divided into pages, each containing 256 words. The upper 20 bits of a virtual address are called the virtual page number (VPN), and the remaining eight bits are the word offset within the page. (See figure 24.) Transfers between main and secondary memory are always done in pages.

Virtual Memory Operation

The 3600 virtual memory scheme is implemented via a combination of Lisp code, microcode, and hardware. The memory-management task is divided into *policies* and *mechanisms.*

Figure 24. 3600 virtual memory mechanisms.



Copyright © 1983 Symbolics, Inc.

Policies are realized in Lisp; these are decisions as to what to page, when to page it, and where to page it to. Mechanisms are realized primarily in microcode and hardware; they implement the policies.

Translating a Virtual Address into a Physical Address

A Lisp pointer contains a virtual address. Before the hardware can reference a Lisp object, the virtual address must be translated into a physical address. A physical address says where in main memory the object is currently residing. If it is not already in main memory, it must either be created or else copied into main memory from secondary memory (disk). Main memory acts as a large cache — the disk is referenced only if the object is not already in main memory. The system attempts to keep the object resident for as long (and only as long) as it is used.

In order to translate quickly and efficiently a virtual address into a 24-bit physical address, the 3600 uses a hierarchy of translation tables. This hierarchy of mapping tables consumes less than 2% of main memory. The levels used are:

- Map Cache
  In the processor, referenced by the hardware. It is a high-speed RAM that can accommodate 8K entries.
- Page Hash Table Cache (PHTC)
  In wired main memory, referenced by the microcode with hardware assist. The size of the PHTC is proportional to the number of main memory pages, and can vary from 4 to 64 Kwords, requiring one word per page frame.
- Page Hash Table (PHT)
  and Main Memory Page Table (MMPT)
  In main memory, referenced by Lisp. The size of both of these tables is proportional to the number of main memory pages, with the PHT being 75% dense and the MMPT 100% dense. Both tables require one word per entry. The PHT and MMPT completely describe all pages in main memory.
- Secondary Memory Page Table (SMPT)
  In main memory. It describes all pages of disk swapping space and dynamically grows as more swapping space is used. It is organized as a B*-tree.

The hardware translates a virtual address into a physical address by checking the map cache for the virtual page number (VPN). If found, the cache yields the physical page number the hardware needs. If the VPN is not in the map cache, the hardware hashes the VPN into a PHTC index, and reads the selected PHTC entry. If it matches the VPN, the map cache is refilled and execution proceeds. Otherwise a *page fault* to Lisp code is generated.[7]

Page faults can occur for two reasons. The first is a cache miss in the map cache and PHTC, in which case the fault handler looks up the page in the PHTC and MMPT, loads the map cache and the PHTC, and returns. The second reason is a requested fault, because of conditions like "I/O in progress," or "The page is not in memory", in which case the handler takes whatever action is required to make the page accessible. If the page is not in main memory, the handler must copy the page from disk into a main memory page. When a page fault gets to this point, it is called a hard fault. A hard fault must do the following:

- Find the virtual page on the disk by looking up the VPN in the SMPT.
- Find an available page frame in main memory. An approximate FIFO (first-in, first-out) pool of available pages is always maintained. When the pool reaches some minimum size, a background process fills it by making the least recently used main memory pages available for reuse. If the page selected for reuse was modified (that is, its contents in main memory were changed so the copy on disk is out-of-date), it must be first copied back to disk prior to its being available for reuse. The background process minimizes this occurrence at fault time by copying modified pages back to disk periodically, especially those eligible for reuse.
- Copy the disk page into the main memory page frame.
- If the area of the virtual page has a "swap-in quantum" specified, the next specified number of pages is copied into available main memory page frames as well. If these

[7]Except for special cases, which are handled in microcode.

prefetched pages are not referenced within some interval and some page frames are needed for reuse, their frames are reused. This minimizes the impact of prefetching unnecessary pages.

- Update the PHT, MMPT, PHTC, and map cache to contain the page just made resident, and forget the previous page that occupied the frame.
- Return from the fault and resume program execution.

Garbage-Collection Mechanisms

In a Lisp environment, storage for Lisp objects is allocated out of an area in virtual memory. Storage must be deallocated and returned automatically when objects are no longer referenced. Garbage-collection routines manage this dynamic storage allocation and deallocation. Garbage collection is the process of finding inaccessible objects and reclaiming their space. This space is then free to be reallocated.

The goal of a garbage-collection algorithm is to reclaim storage quickly and with a minimum of overhead. Conventional garbage-collection schemes are computationally costly and time-consuming, since they involve reading through the entire address space — a major task when a large address space is used. This is done in order to prove that nowhere in the address space do references to the storage being considered for reclamation exist.

Hardware-assisted Garbage Collection

Garbage collection on the 3600 is incremental, being run when necessary in the background. The design of the 3600 includes unique features for hardware assistance to the garbage-collection algorithms, which greatly simplify and speed up the process. These hardware features are used to mark parts of memory to be included in the garbage-collection process, leaving the rest of memory untouched. The hardware features include:

- Type fields
  Distinguishes memory words containing pointers from those containing numbers
- Page Tags
  Indicates pages containing pointers to temporary space

↗ Multiword read instructions
Speeds up the memory scanning

The 2-bit type field inserted into all data words by the hardware simplifies garbage collection. This field indicates whether or not the word contains a *pointer* — that is, a reference to a word in virtual memory.

Each physical page of memory has a page-tag bit. This is set by the 3600 hardware when a pointer to a temporary space is written into any location in that page. When the garbage-collector algorithm wants to reclaim some temporary space, it scans the page-tag bits in all the pages. Since the page-tag memory is small relative to the size of virtual memory, it can be scanned rapidly (about two ms per Mword of main memory that it describes). For all pages with the page-tag bit set, the garbage collector scans all words in that page, looking for pointers to "condemned" temporary space. For each such pointer it copies out the object pointed to and adjusts the pointer so that it points at the copy. The garbage collector also scans the stacks.

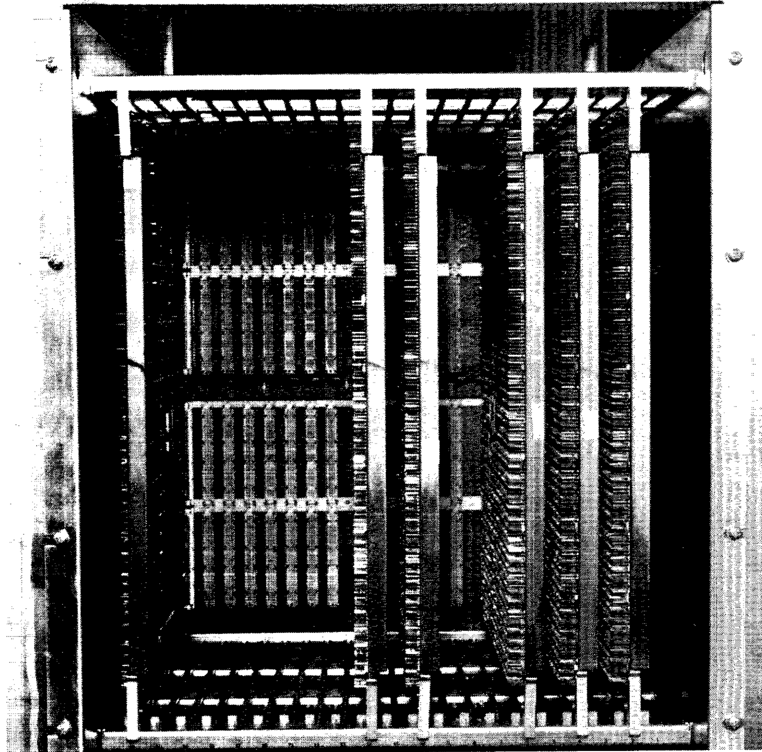
Multiword read operations speed up the garbage collection by fetching several words at a time to the processor. This makes the page-scanning faster.

The virtual memory software assists garbage collection with another mechanism. When a page with its page-tag bit set is written to disk, the 3600 paging software scans through the contents of the page to see what it is pointing at. The software maintains a table recording the swapped-out pages which contain pointers to temporary spaces. Since the garbage collector checks this table, it can tell which pages contain such pointers. This knowledge is used to improve the efficiency of the garbage-collection process. Only pages that actually contain pointers to the condemned space need to be read in from the disk before the space can be reclaimed. In this way, inefficient interaction between the garbage collector and the virtual memory is avoided. The garbage collector does not greatly increase the page-swapping activity of a user's program.

The L Bus

The L bus backplane connects the processor to memory and to high-speed peripherals (see Figure 25). Peripherals on the L bus include the disk, network, and TV controllers, as well as the front-end processor (FEP). The address paths of the L bus are 24 bits wide, and the data paths are 44 bits wide, including 36 bits for data, seven bits for error correction, and one bit spare. The L bus is capable of transferring one word per cycle at peak performance (approximately 20 Mbyte/sec).

Figure 25. View of the L bus backplane.



A central memory control unit manages the state of the L bus and arbitrates requests from the 3600 processor, the IFU, the FEP, or other DMA devices. It also performs error detection and correction.

As an example of L bus operation, a normal memory read cycle includes three phases.

- Request
  The processor or the FEP selects the memory card from which to read (address request).
- Active
  The memory card accesses the data; the data are strobed to an output latch at the end of the cycle.
- Data
  The memory card drives the data onto the bus; a new request cycle can be started.

A normal write operation includes two phases.

- Request
  The processor or the FEP selects the memory card to which to write.
- Active
  The processor or the FEP drives the data onto the bus.

Block-Mode Operation

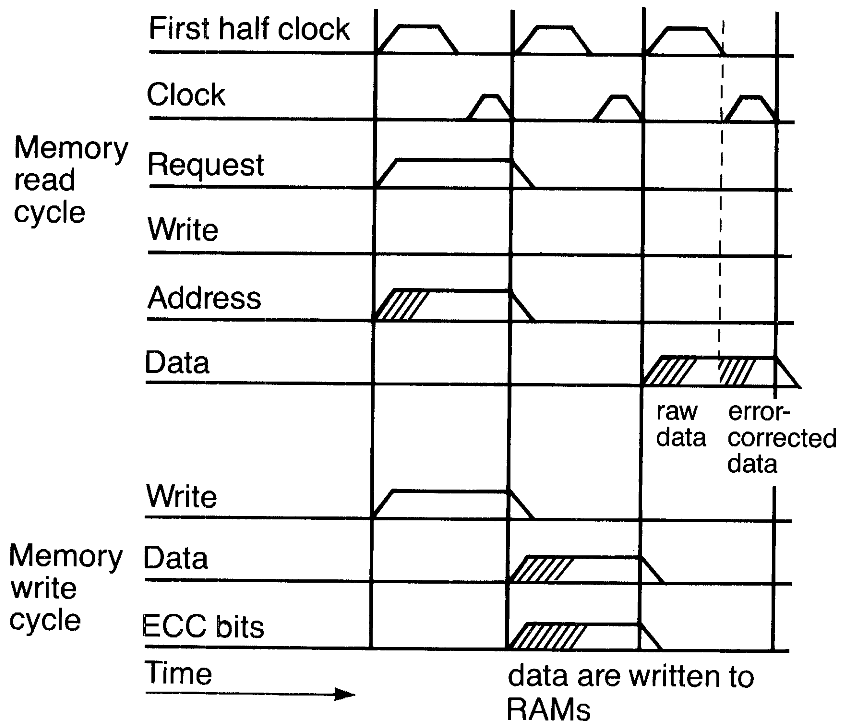
The L bus logic facilitates block-mode memory operations. In block-mode operations, successive memory locations are accessed on each cycle. This is especially useful in copying or searching procedures. On the 3600, block-mode operations apply only to sequential addresses, since the memory boards are internally interleaved.

Parallel pipelining techniques are used in the L bus logic to overlap several bus requests. For example, on block-mode memory writes, an address may be requested while a separate data transfer takes place. On block-mode memory reads, a new request is initiated every L bus cycle.

Direct Memory Access (DMA) Operation

A modified L bus memory cycle is used for DMA operations by devices on the bus. In a DMA output operation (from memory to a device), as in all memory read operations, the data from memory are routed to the ECC logic. Then the data are shipped to the appropriate DMA device, for example, FEP, disk controller, network controller.

Figure 26. L bus timing.



Both *microcode-mediated DMA* and *device-controlled DMA* are possible. In microcode-mediated DMA, a microcode task supplies the address, while the DMA device supplies or receives the data. The design of the microtasking hardware ensures that using microcode-mediated DMA for the disk imposes a peak load of only 12% of the processor.

In device-controlled DMA, the device supplies the address; processor microcode is not involved. Device-controlled DMA is used by the FEP and by other devices that need fast nonsequential access to memory on the L bus.

The L Bus Clock

Main memory and 3600 processor operations are synchronous with the L bus clock, as are all L bus operations. The clock rate is roughly 5 MHz, but the exact cycle can be tuned by the microcode. A field in the microcode allows different speed instructions for different purposes. For example, fast instructions need not wait the long clock cycle needed by slower instructions. When the processor takes a trap, the clock cycle is stretched so that a trap-handler microinstruction can be fetched.

symbolics
"What a Win !!"
"What a Win !!"

3600 Hardware: I/O Systems

This chapter discusses the major input/output systems associated with the 3600, including the front-end processor (FEP), serial lines, the Spy bus, the console, and the digital audio output system.

The FEP and MULTIBUS

The 3600 includes an MC68000-based FEP which serves several functions. During normal operation, the FEP controls the low- and medium-speed input/output (I/O) devices, logs errors, and initiates recovery procedures if necessary. The use of the FEP drastically reduces the real-time response requirements imposed directly on the 3600 processor. Devices such as the mouse, keyboard, serial lines, parallel port, and cartridge tape are connected to the 3600 via the FEP.

The FEP is supplied with 128 Kbytes of RAM (random access memory) and 64 Kbytes of EPROM (electrically programmable read only memory). An optional IEEE-796 (MULTIBUS) card cage can be used to attach commercially available MULTIBUS peripherals to the 3600 via the FEP. The FEP contains the MULTIBUS memory map within its address space.

Four programmable serial lines (including the AM1200 modem) as well as MULTIBUS DMA data are routed through the FEP. MULTIBUS devices can perform DMA operations directly to the 3600's L bus. DMA operations from the FEP to 3600 memory can be carried out at a rate of more than 1 Mbyte per second.

Serial Lines

Four serial lines are connected to the FEP. Two are high-speed and two are low-speed. Each one can be used either synchronously or asynchronously. One high-speed line is always dedicated to the 3600 console. One low-speed line must be dedicated to an AM1200 modem, if it is present. The transmission rate of the low-speed lines is programmable, up to 19.2K bps. The available high-speed line is capable of speeds up to 1M bps. The nonconsole lines are terminated using standard 25-pin D connectors (EIA RS-232 compatible).

MULTIBUS Interrupts

The FEP processes real-time interrupts from the MULTIBUS. After receiving an interrupt, the FEP traps to the appropriate interrupt handler. Users can write additional handlers in LIL, the programming language for the FEP (see page 50.) The handler might process the interrupt on its own or relay a message to the 3600 processor or to some other device.

The LIL interrupt handler can intercommunicate freely with Lisp code running in the 3600 processor, using shared memory and real-time process wakeups.

Bootstrap Loading the 3600

The FEP performs the following operations in bootstrap loading the 3600:

- Runs diagnostic checks on the 3600 processor, display screen, and disk
- Loads bootstrap program into the FEP memory
- Gets the microcode from the FEP file system and puts it in the 3600
- Reads virtual memory code into 3600 memory
- Sets up machine-configuration table
- Starts the 3600 processor

If the 3600 fails any of the diagnostic tests, the FEP notifies the user via the console terminal or the 12-character display on the front panel of the processor, depending upon the nature of the failure.

Hardware Error Handling

The FEP reports hardware error signals from the 3600 processor. If the errors come from hardware failures detected by consistency checks (for example, parity errors in the internal memories), the 3600 processor stops. At this point the FEP directly tests the hardware and either continues the processor or notifies the user. If the error signals are generated by microcode or Lisp interacting with the hardware, the FEP records the error. These are typically disk or memory errors, not pure software errors like an unbound variable.

Periodically, the 3600 requests information from the FEP and

records it on the disk, to be used by the maintenance personnel. Since the FEP always has the most recent error information, it is possible to retrieve it when the rest of the machine crashes. This is especially useful when a recent hardware malfunction causes a crash. Since the error information is preserved, it can be recovered when the 3600 processor is revived.

Running Diagnostics from the FEP

When the 3600 is not running, the FEP provides diagnostic and microcode debugging tools. Simple diagnostics are stored in an EPROM within the FEP, along with code to run the network and the disk. More complex diagnostics are stored on disk or can be loaded from the network. The FEP uses the Spy bus to test the internals of the 3600 processor.

For debugging hardware problems, it is possible to connect to the FEP from another console on the Ethernet or by dialing in from a remote console. Should more sophisticated diagnostics be needed, these can be loaded from the disk or over the network connection.

The Spy bus

The diagnostic interface to the 3600 includes the Spy bus. This is an 8-bit wide bus which can be used to read from and write to various portions of the 3600 processor. The readable locations in the processor allow the FEP to spy on the operation of the 3600 processor, hence the name Spy bus. For example, using the Spy bus, the FEP can force the 3600 processor to execute microinstructions for diagnostic purposes. The FEP can examine data paths and registers to pinpoint the source of a hardware failure.

When diagnostics are not running, the FEP uses the Spy bus as a special channel to certain DMA devices. Normally, the FEP uses the Spy bus to receive a copy of all incoming Ethernet packets. It can also set up transfers to the Ethernet and read from the disk via the Spy bus.

The FEP File System

The FEP file system resides on the main disk. It contains all the files required to boot and diagnose the 3600 processor. It also acts as the first level of disk organization, since it contains files which are not part of the local file system, such as the paging area, microcode files, and the local file system itself. Every page on the disk is contained in the FEP file system.

Both the FEP file system and the local file system have their place. While the local file system is designed for speed and flexibility, the FEP file system is designed primarily for simplicity and survivability.

The FEP file system is integrated into the normal pathname host system. To the 3600 software, it looks very similar to the local and remote file systems. For example, the normal file-copying operations can be used to copy FEP file-system files from machine to machine.

The NanoFEP

The NanoFEP is the front-end to the front-end processor. This tiny microcomputer is charged with some small but important tasks. Specifically, the NanoFEP performs the following operations:

- Boots the FEP
- Manages the front-panel lights on the processor cabinet
- Monitors the temperature inside the processor cabinet
- Sets the time-of-day clock

Based upon an Intel 8749 microcomputer chip, the NanoFEP has a total of 50 bytes of battery-backed RAM.

The 3600 Console

The standard 3600 console includes a high-resolution (1150 x 900 pixel) black-and-white monitor, an MC68000 microprocessor subsystem,[8] an 88-key alphanumeric keyboard, a mouse, and a headphone jack.

---

[8] This is a second MC68000, in addition to the MC68000 in the FEP.

Within the console are interfaces for the following devices:
- Alphanumeric keyboard
- Mouse
- Digital audio output system

The aspect ratio of the console display is 1.27 horizontal to vertical. For viewing visibility, you can tilt or turn the console. The screen is refreshed at 60 Hz (noninterlaced) on a P104 phosphor. The bit map is a bit-per-pixel video generator, consisting of a raster memory, a rasterizer, and a sync generator. The bit map appears as memory on the L bus.

Communication from the console to the 3600 is processed through the FEP. Communication between the FEP and the console processor is based on a 600K bps bidirectional link over a serial line.

In the standard console configuration, the digital audio and the video on the monitor are driven directly by the 3600 processor, not by the MC68000 in the console. Via its interface, the MC68000 in the console handles all other communications, such as keyboard and mouse.

Digital Audio Output System

A digital audio output system (see figure 27) will be standard on the 3600 console. The console presently offers only monaural conversion. However, the 3600 processor is capable of providing two 16-bit audio channels.

Audio output is a function of a microtask, driven by a microtask wakeup signal sent from the input/output board (IOB). Stereo audio samples are 32 bits in length. The most significant 16 bits are the left channel, while the least significant 16 bits are the right channel. With 16-bit stereo samples, approximately 2% of the processor's time is used in outputting precomputed audio samples, at a 50 KHz sample rate.

The digital audio output from the processor is double-buffered on the IOB. Since the audio shift register takes 20 microseconds to shift out a full stereo sample to the Manchester-encoding logic, the processor has this much time to forward the next sample to the buffer register.

The serial audio data from the shift register includes 40 bits which facilitate decoding at the console end. The bit stream format is as follows:

```
 1  start bit
16  bits of sample
 1  channel bit (L)
 2  stop bits

 1  start bit
16  bits of sample
 1  channel bit (R)
 2  stop bits
-------------------
40  bits total for a stereo sample
```

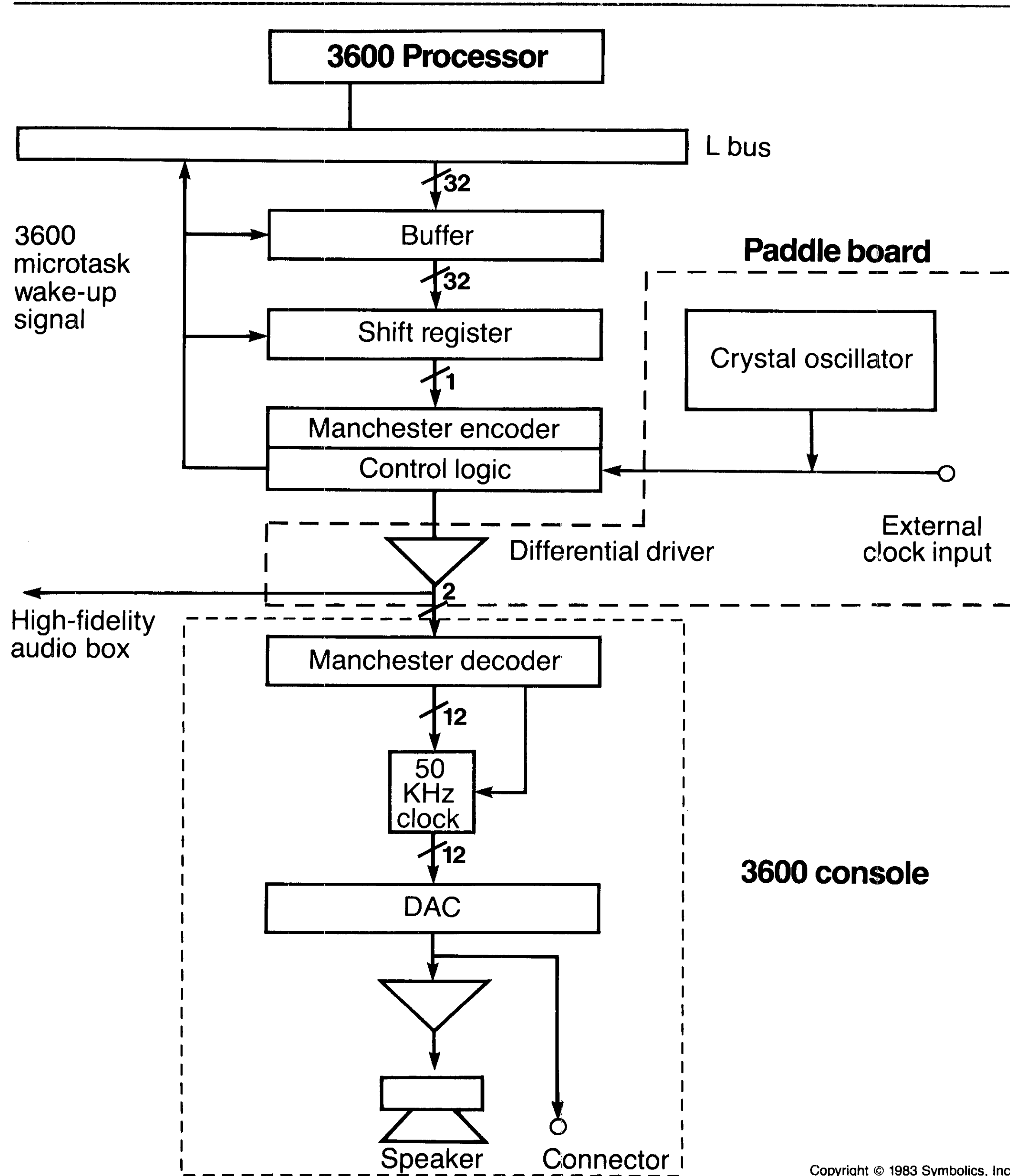
All 40 of the bits must be transmitted in one sample time. For a 50-KHz sampling frequency, this yields a 2-MHz bit rate. The bit-clock comes from a plug-in crystal oscillator on the I/O paddle card. The frequency of this oscillator is 80 times the desired audio sampling frequency (twice the bit rate). If connection to external digital audio equipment is desired, the oscillator can be left out and the board will accept an external clock via a card-edge BNC connector.

Manchester encoding allows audio data to be transmitted along with the bit-clock down a single serial line to the console. At the console end, the serial data stream is decoded and the clock is extracted. The least significant 12 bits of a stereo sample (the right channel) are sent to a 12-bit digital-to-analog converter (DAC). The analog output from the DAC is amplified and sent to a small speaker in the console. A connector on the console is also provided to tap the converted analog audio signal.

Disk Controller

The 3600 disk controller is closely linked via the L bus to the 3600 processor. With this controller, any storage module drive (SMD) unit can be attached to the 3600. The controller has error correction circuitry that allows the correction of burst errors of up to 11 consecutive bits on a disk page. The disk controller hardware handles rotational position sensing and

Figure 27. Basic digital audio output system.



Copyright © 1983 Symbolics, Inc.

overlapped seeks. To simplify the controller logic, memory accesses and disk header comparisons are performed in processor microcode, as a separate microtask. This microcode assist allows page-chaining so that more than one page can be transferred in a single request.

3600 Hardware: Packaging and Specifications

Processor and Console Cabinets

The basic 3600 system is packaged in two cabinets: a processor cabinet and a console cabinet. The processor cabinet contains the processor, FEP, and memory boards. In addition, it holds a cartridge tape drive, a disk drive, and power supplies.

The console cabinet is connected to the processor cabinet via a cable up to 60 meters in length. The Ethernet transceiver is mounted on a local network coaxial cable and must be within six meters of the processor cabinet.

Additional cabinets for the color display, additional disks, tape drives, and the MULTIBUS card cage are optional.

The 3600 processor uses Schottky TTL circuitry, with 10K and 100K ECL (emitter-coupled logic) along certain critical paths.

Reserved, color-coded slots are provided on the L bus backplane for several boards:

- Data-path and arithmetic-logic-unit board (DP)
- Sequencer board (SQ)
- Front-end processor board (FEP)
- Instruction fetch unit and memory-controller board (IFU)

The rest of the backplane is undedicated, with seven free L bus slots.[9] The input/output board can go into any undedicated slot. Plugging a memory board into an undedicated slot sets the address of that board. For diagnostic purposes, the FEP can always tell which board is plugged into what slot; it can even tell the serial number and ECO (engineering change order) level of the board.

No internal cables are used in the 3600. All board-level interconnections are accomplished through the backplane. A 16-meter external cable is provided with the standard system for connecting the console to the 3600 processor cabinet. Longer cables up to 60 meters are available as options.

---

[9]Future systems will have 14 free L bus slots.

Figure 28. Front view of the board layout on the backplane. An additional cabinet is provided for memory expansion beyond that shown here.

(Optional color controller) or Memory	(Optional color memory plane) or Memory	(Optional color memory plane) or Memory	(Optional color memory plane) or Memory	(Optional color memory plane) or Memory	Memory	Memory	Memory	I/O controller	Front-end processor		Memory controller	Sequencer	Data path
L	L	L	L	L	L	L	L	L					

Electrical Specifications

The processor requires 110 volts AC at 30 amperes and consumes approximately 2000 watts. The console requires 110 volts AC at three amperes. It consumes approximately 1100 watts. 60-Hz operation is standard, while 50-Hz operation is available.

Four switching power supplies exist on a basic 3600 system: one for the disk, two for the processor logic, and one for the NanoFEP. All of the voltage regulators are embedded in the power supplies.

Environmental Specifications and Requirements

The 3600 operates reliably between the temperatures of 32 to 90 degrees Fahrenheit (0 to 25 degrees centigrade), with a relative humidity of from 15 to 80% (noncondensing). The processor dissipates 6800 BTUs of heat per hour. A thermostat inside the processor controls the speed of the cooling fans.

Physical Dimensions and Weights

The dimensions and weights of the 3600 hardware components are as follows:

- Processor cabinet
  - 172.5 cm (69 in) high
  - 56.25 cm (22.5 in) wide
  - 80 cm (32 in) deep
  - 387 kg (850 lbs) in weight
- Console
  - 38 cm (15.2in) high
  - 42.5 cm (17 in) wide
  - 29.2 cm (11.68 in) deep
  - 18.2 kg (40 lbs) in weight
- Optional color display
  - 44 cm (17.6 in) high
  - 44 cm (17.6 in) wide
  - 48 cm (19.2 in) deep
  - 46 kg (101 lbs) in weight

symbolics

3600 Hardware: Peripherals

The 3600 is designed to support three classes of peripherals:

- L bus
  Disks, color display system, Ethernet interface
- MULTIBUS
  Tape drives, modem
- FEP
  Laser printer, serial lines

L bus peripherals are accessed very rapidly, while the Ethernet peripherals are serviced the slowest. This section discusses each class of peripheral in turn.

Disk Systems

The 3600 uses a 169-Mbyte unformatted fixed-media Winchester disk drive as its basic paging and local storage system. The average positioning time is 27 ms. The maximum seek time of the disk is 55 ms. The transfer rate is 1.012-Mbytes/sec. The disk is mounted inside the processor cabinet.

As an option, fast-access DD470 fixed-media Winchester disks can be added to the basic system. These disks hold 474 Mbytes unformatted. DD470s have an average positioning time of 18 ms, a maximum seek time of 35 ms, with a transfer rate of 1.859 Mbytes/sec. One drive will also fit in the processor cabinet, replacing the standard disk.

The maximum disk load on a 3600 with a single input/output board is 1.896 Gbytes (four 474-Mbyte drives). With another input/output board (one L bus slot), this capacity can be doubled. Extra disk drives are housed in separate cabinets.

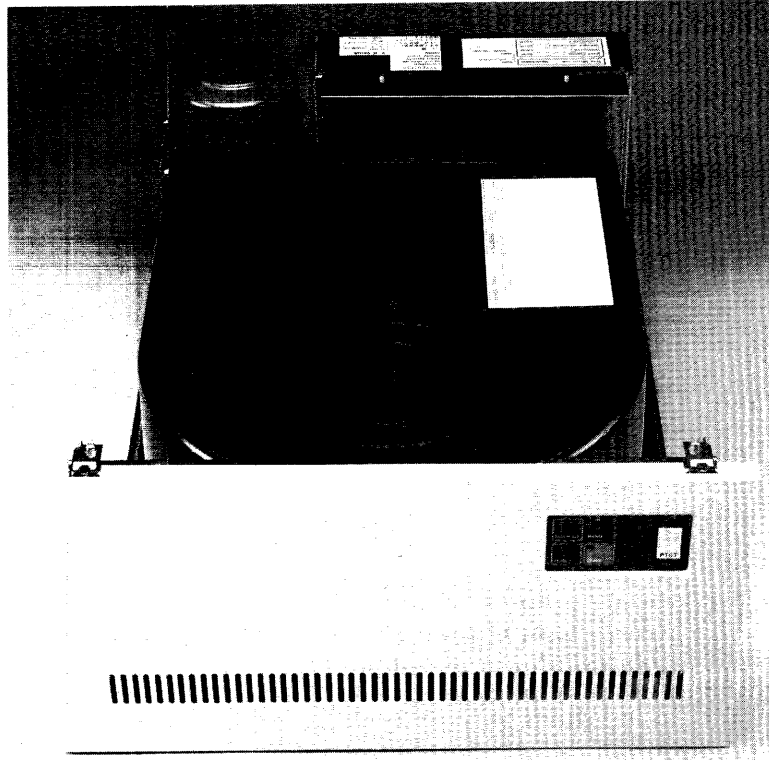
Applications that require removable disk media can be configured with a free-standing 300-Mbyte storage module drive.

Ethernet Interface

The 3600 processor and the network controller (part of the input/output board) work closely to provide support for the full 10-Mbit/sec Ethernet protocol. The processor and network controller communicate via the L bus. The network controller also receives a microtask assist from the processor.

Ethernet is a local area network transport mechanism consisting

Figure 29. The 169-Mbyte disk drive.



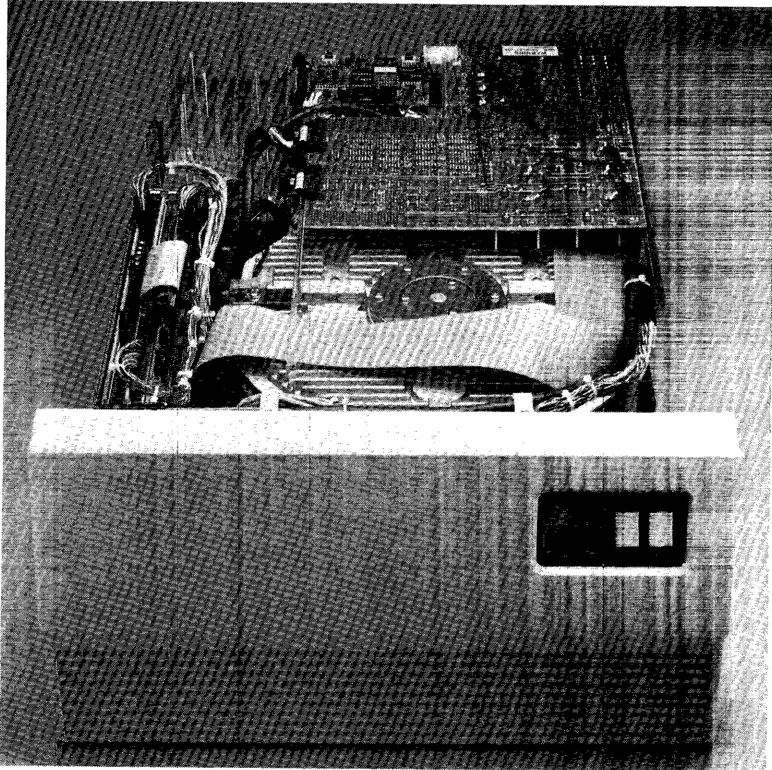
of a packet bus that uses statistical arbitration. The physical connection between Ethernet nodes is a standard single-strand coaxial cable, tapped by a transceiver. Up to 100 Ethernet stations can be interconnected on a 1000-meter cable. For compatibility with existing installations, Chaosnet local area network support is available via a gateway processor.

Color Display System

The CD1000 color display system option is a controller and memory system that provides flexible, programmable, high-resolution color graphics.

The color display system is designed to support graphics applications in the fields of computer-aided-design (CAD), image processing, and animation.

Figure 30. The 474-Mbyte disk drive.



The minimum configuration of this 1280 x 1024 pixel display consists of a color controller board, one color memory (CMEM) board, and a video paddle card. The paddle card contains all the connectors and electrical interface hardware. This configuration gives the user two million 8-bit pixels. Ten bits control the output of each color gun — red (R), green (G), and blue (B) — allowing the potential for one billion colors. Three coaxial cables allow operation of the CD1000 up to 16 meters from the processor.

Standard features of the CD1000 system include:

- 64 x 64 x 2 writable nondestructive cursor overlay (the 2 bits determine the choice of color map)
- 2,097,152 pixels stored in memory at all pixel depths
- Microcoded sync generator with writable control store

(allows programming of any video format up to 1280 x 1024)

- Two color map modes
  - Pseudocolor: 256 colors (eight bits)
  - Full color: 256 levels each of RGB (16.8 million colors provided by 24 bits)
- Zoom by integer ratios independently in the *x* and *y* dimensions, by up to 255
- Pan to the pixel independently in *x* and *y*
- Line index table that maps data in the color memory into a logical raster, each line of which can be independently panned and zoomed by line-attribute fields in the table
- Color memory mapped directly into 3600 address space
- Pixel and plane write-mask registers mask off pixels and/or planes during memory operations
- Pixel writing at up to 1.7 Gbits/sec in Fill addressing mode

Color Memory Addressing Modes

The memory topology of the 3600 color frame buffers provides for four addressing modes:

- Pixel mode
  Accesses one pixel per 32-bit word.
- Packed mode
  Bytes of four or two adjacent pixels are packed into one 32-bit word.
- Plane mode
  Accesses one plane of 32 adjacent pixels.
- Fill mode
  Fills 32 adjacent pixels with the value of the color-fill register in one memory-write cycle. It reads memory like the Plane mode under control of a special field in the control register.

Each of the four modes can be seen as a different topological mapping of the physical bits in the color memory into a virtual block of address space. In other words, the same physical memory is mapped into four different blocks of the machine's virtual memory. Hence, selection of a color addressing mode is determined by the address.

In Plane addressing mode, the color memory is addressed as a stack of 32 planes. The first eight planes are red, the next eight

are green, the next eight are blue, and the last eight are overlay planes. A 32-bit word at location 0 is mapped into the first 32 bits of the first red plane, along the *x*-axis (see figure 31). Plane mode is useful in CAD applications where multiple layers of an image are superimposed, such as in VLSI design.

In Pixel addressing mode, memory is viewed as a three-dimensional block of cells. A 32-bit word at location 0 is mapped along the *z*-axis of the block, with eight bits of red, eight bits of green, eight bits of blue, and eight bits of overlay (see figure 32). Pixel mode is useful for image processing and other applications where the pixel is the important addressable unit.

In Packed addressing mode, memory is also viewed as a block, but in a different configuration. A 32-bit word at location 0 is mapped into eight bits of color at *x*-location 0, eight bits of color at *x*-location 1, eight bits of color at *x*-location 2, and eight bits of color at *x*-location 3 (see figure 33). Packed mode operation is especially efficient with an eight-bit-deep color memory, since one write operation covers four full-color pixels.

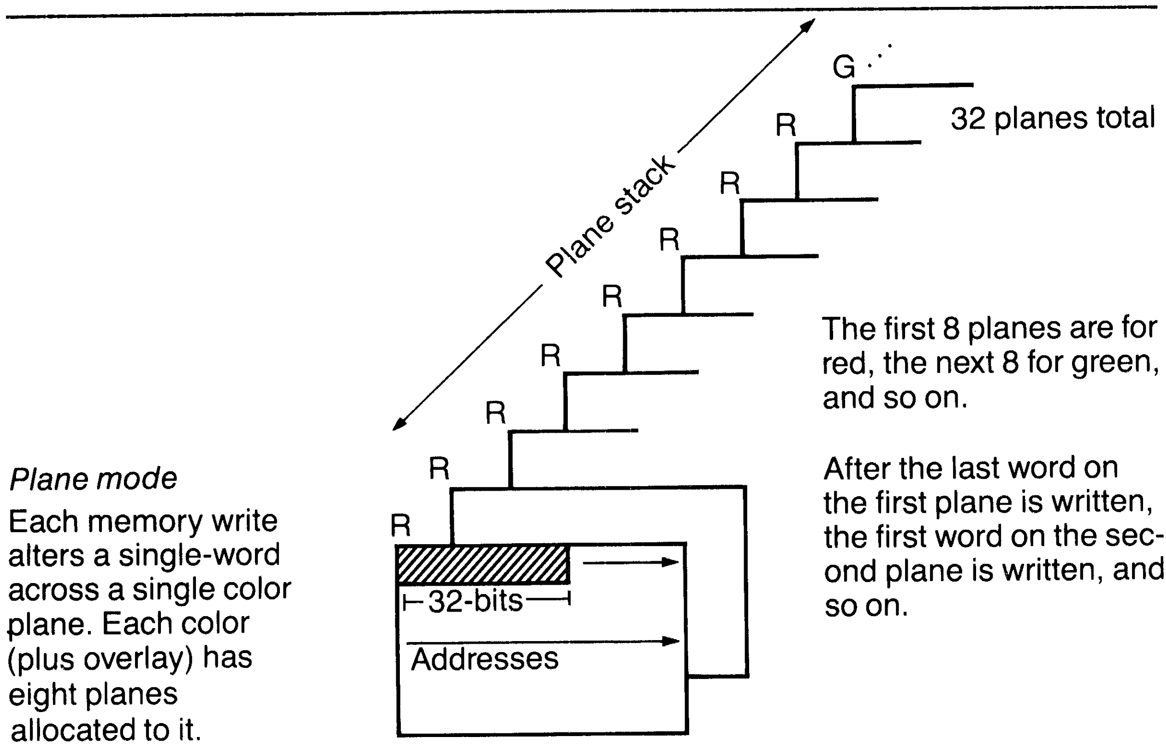
In Fill addressing mode, special control logic performs a two-stage mapping operation using the color parameter register as a data source. The color parameter register is a single 32-bit register which stores a color (see figure 34). Subsequent memory writes use data words as collections of write-enable bits. The color that is written into an enabled pixel is that stored in the color parameter register. Pixels that are not enabled are not altered. In Fill mode, graphic effects such as half-tones, stipple patterns, and characters can be written extremely quickly. Since the 32-bit write operation is mapped through the control card with up to a 32-bit color value, one memory write in Fill mode generates effectively 32 x 32 (1024) bits in a single 600-ns write cycle. Hence, graphically intensive operations, such as area fills, can be carried out at a rate of 1.7 Gbits/sec.

Color Display System Options

The standard color display system is supplied with eight bits per pixel. One, two, or three additional color memory cards can be plugged in for depth expansion to 16, 24, or 32 bits per pixel, respectively.

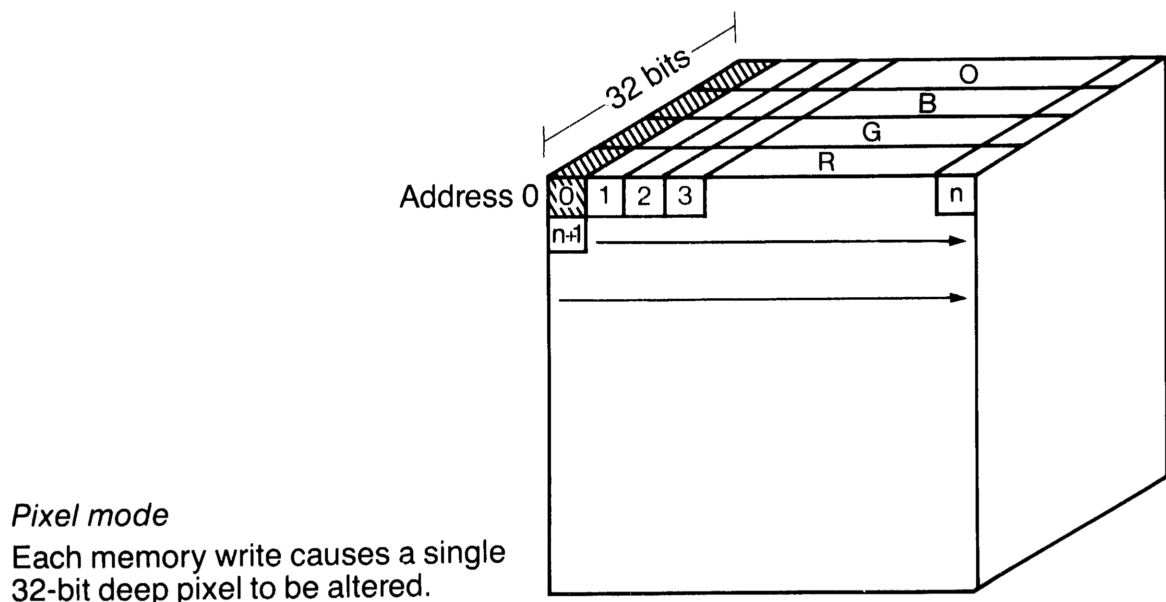
Figure 31. Color memory topology in the Plane addressing mode.

Figure 32. Color memory topology in the Pixel addressing mode.



Plane mode

Each memory write alters a single-word across a single color plane. Each color (plus overlay) has eight planes allocated to it.

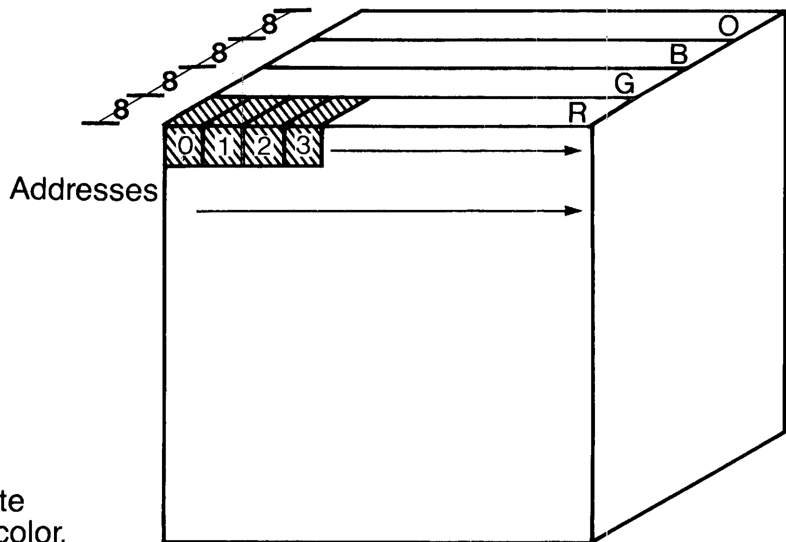


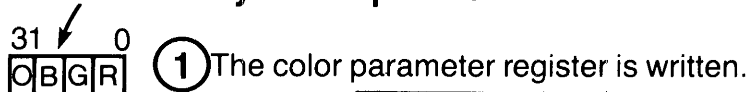
Pixel mode

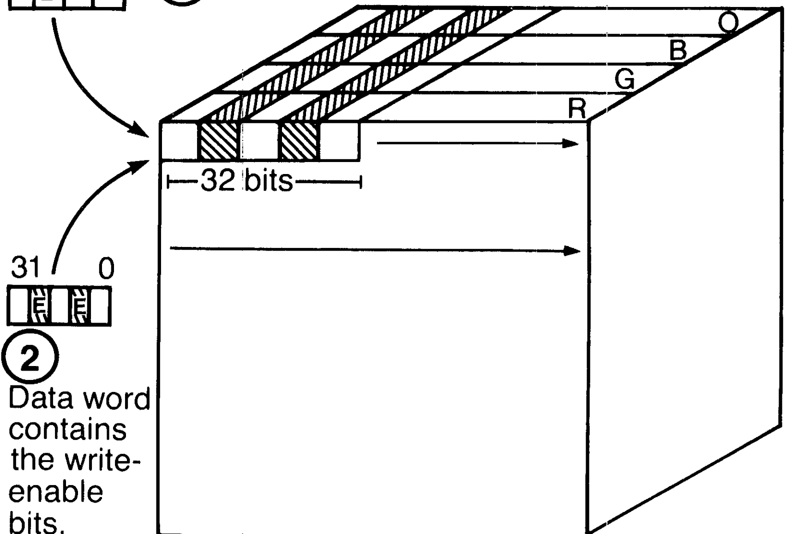
Each memory write causes a single 32-bit deep pixel to be altered.

Figure 33. Color memory topology in the Packed addressing mode.

Figure 34. Color memory topology in the Fill addressing mode.







Tape Drives

Symbolics supplies several MULTIBUS-compatible tape drives for backup, distribution, and compatibility with other systems.

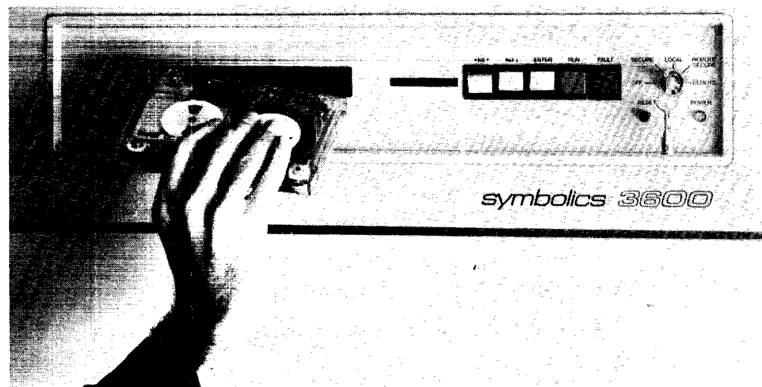


Figure 35. The TD20 cartridge tape drive.

Each 3600 site is supplied with an embedded TD20 20-Mbyte, quarter-inch cartridge tape drive (see figure 35) for receiving software updates from Symbolics. The TD20 can also be used for backup of the disk. Each cartridge holds up to 20 Mbytes of data.

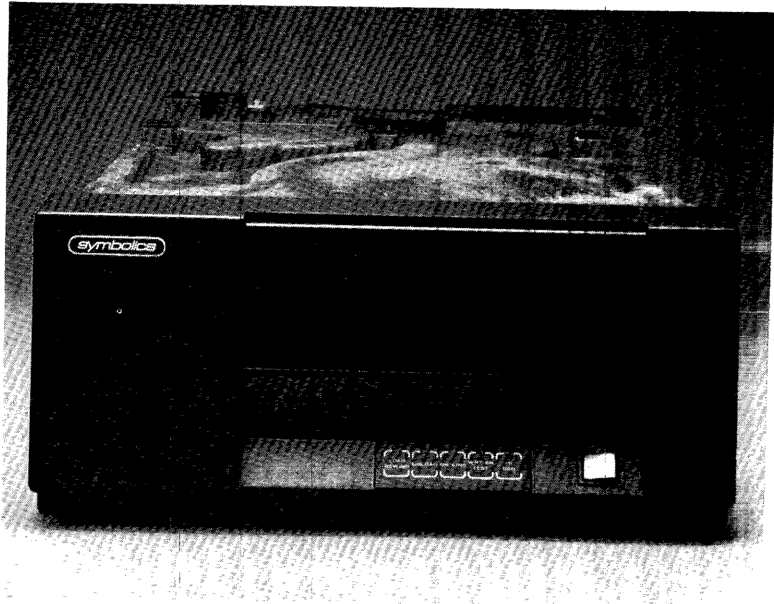
As an option, a 3600 configuration can include a TD80 tape drive (see figure 36). The TD80 is an industry-standard 9-track, 1600 bpi, 25 ips, half-inch tape drive. The TD80 also performs in a 3200 bpi, 90 ips streaming mode. The TD80 may be housed in a standalone cabinet or in a DD470 disk drive cabinet. The TD80 and other tape drives can be attached to the 3600 via the MULTIBUS interface.

Laser Graphics Printer

The Symbolics LGP-1 laser graphics printer is a table-top laser-beam printer with an MC68000-based controller. The LGP-1 is designed to produce high-quality graphic documents on standard 8.5 x 11 inch size paper.

The LGP-1 combines precision optics, semiconductor laser beam recording, and electrophotographic copier technology to produce documents and graphics at a resolution of 240 dots/inch

Figure 36. The TD80 streaming tape drive.



horizontal by 240 dots/inch vertical. This document was produced from masters printed on a Symbolics LGP-1.

Features of the LGP-1 include:
- Nonimpact, quiet printing
- Variable-width characters, multiple fonts, and multiple font sizes
- Portrait- and landscape-aspect printing
- Tektronix compatibility mode
- 9.8 letter-sized pages per minute
- Screen images from bit-mapped displays
- Minimum processing time on host computer
- Power-saving automatic shutoff
- Dynamic font loading
- Support for Troff and TEX formatter output
- Industry-standard interfaces (RS-232C asynchronous and DEC DR11-C compatible parallel)

The LGP-1's controller is driven by an MC68000 microprocessor with 1 Mbyte of memory in the basic configuration. Standard interfaces can be connected to the LGP-1. These include (with their transfer rates):
- The RS-232C serial line interface

Up to 19,200 bps

- The 8-bit parallel interface compatible with the 3600
  Up to 1 Mbyte/sec
- The 16-bit parallel interface compatible with DEC DR-11C
  Up to 2 Mbytes/sec

The supplied software of the LGP-1 allows it to work as a line printer or graphics printer. The unit also runs standalone diagnostics and prints test patterns. In graphics mode, the LGP-1 interprets commands to load fonts, draw characters from fonts, and draw lines, rectangles, and pixel arrays. Commands to magnify images are also available. "Filter" programs, to convert information from one graphic format to another, are not needed.

The LGP-1 Laser Graphics Printer accepts and prints data prepared for an ASCII printer, and the Tektronix graphic display. A native-mode format is available for maximum flexibility.

For use with UNIX systems, four TROFF fonts (regular, italic, bold, and mathematical symbols) are supplied, along with line printer fonts (normal and rotated), and all the standard Computer Modern fonts often used with the TEX formatter. In addition, the font editor supplied as part of the 3600 software package allows users to create fonts.

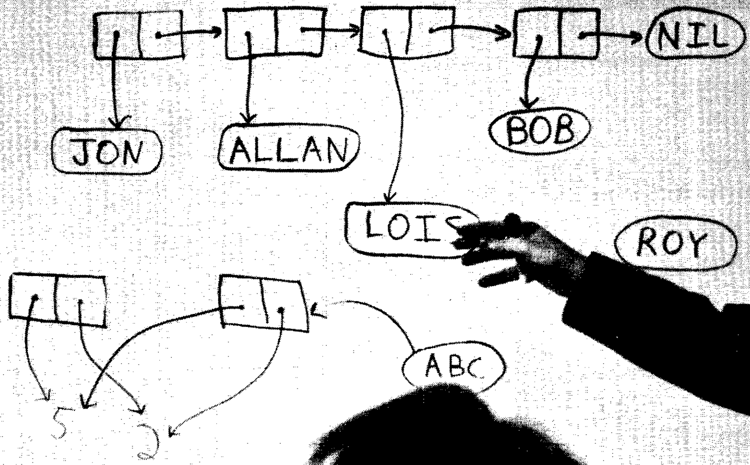
3600 Technical Communication

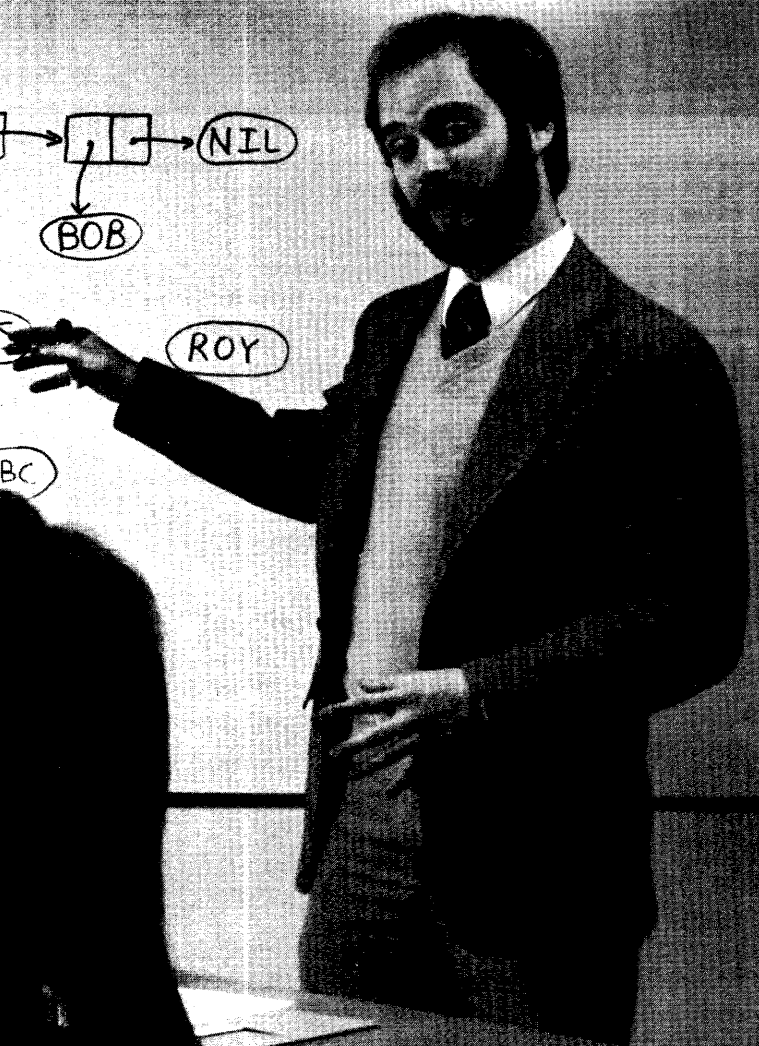
Symbolics provides technical information in many forms, including printed manuals, training courses, and online documentation. Key components of the 3600 technical communication package are:

- Introduction to Lisp and Lisp Machine Programming
  Experienced programmers with no Lisp experience learn to program and operate the Lisp Machine in this two-week intensive course. Each student develops a program project, using many advanced features of the Symbolics system. Students also learn to read and modify large programs. Examples are drawn from actual Lisp Machine system code.
- Concepts and Techniques documents
  These documents present an overview of a topic for experienced programmers who are new to the Lisp Machine. It draws on the rich contextual background of these users in presenting both conceptual architecture and specific features. From these documents, users derive a broad enough understanding to feel comfortable poking around on the machine, trying things out and learning on their own. An example is *Zmail Concepts and Techniques.*
- Advanced Programming Seminars
  Lisp Machine programmers learn advanced concepts and techniques from seminars given by senior members of the Symbolics technical staff. These seminars vary according to the needs of the students. In the past they have covered such topics as program design, object-oriented programming, and networks.
- Reference manuals
  These documents are organized by subsystem, program, or other software or hardware entity. They cover a given topic in complete detail. The presentation of the material reflects the organization of the code. Examples are the *Lisp Machine Manual* and *Signalling and Handling Conditions.*
- Survey documents
  These documents are summary collections of information from many sources. Each survey addresses one topic or function in detail, but it draws source material from many subsystems where possible and practical. Examples are the *Lisp Machine Summary* and *Program Development Help Facilities.*

- Update documents
  These documents present new features or changes to existing software based on new releases of the system software. An example is *Release 4.0 Release Notes.*
- Procedural manuals
  These documents give step-by-step instructions for accomplishing specific tasks. The level of complexity varies, depending on the topic. An example is the *Software Installation Guide.*
- Hardware documentation
  Hardware schematics describe the following parts of the 3600 system:
  - Datapath (DP) (the processor)
  - Sequencer (SQ) board
  - Front-end processor (FEP) board
  - Memory controller (MC) board
  - Input/output board (IOB)
- Online documentation
  Standard online documentation facilities are built into the software system. These include the following:
  - The status line indicates what state the machine is in.
  - The mouse documentation line describes the effect of pressing the mouse buttons in a given context.
  - Help with a particular application or context is accessible by pressing the HELP key on the keyboard.
  - Summary documentation and function argument lists are available on single-keystroke commands.

LISP
Release Notes for System 78
System 210 Release Notes
Release 4.0 Release Notes
Signalling and Handling Conditions
Zmail Concepts and Techniques
symbolics
Software Installation Guide
symbolics
Program Development Help Facilities


NIL
JON
ALLAN
BOB
LOIS
ROY
ABC
5
2



Glossary

- Address Space
  The programs and data used in computers are stored in addressable memory locations. The size of the total available memory is called the address space of the computer. This size can be expressed as the number of bits it takes to hold the largest address. The 28-bit virtual address space of the 3600 refers to a total of 256 million addresses. Each addressable unit (word) holds 36 bits.
- Areas
  Storage in the 3600 memory is divided into areas. Areas are intended to give users control over the paging behavior of their programs. Areas can also be used to "fence off" a portion of memory from the garbage-collector.
- **bitblt**
  This abbreviation stands for "bit boundary block transfer" (also known as RasterOp on some systems). On the 3600, **bitblt** is a fast microcoded array operation that can be used to move a rectangular portion of the screen from one place to another. **bitblt** can also perform Boolean operations and repetitions. It is used for painting characters, drawing lines, scrolling an image across a window, moving an image onto the screen from somewhere in memory, and saving the undisplayed portion of an image (for example, a corner of a window covered up by another window).
- Bit-mapped display
  This is a display in which one memory cell is allocated for each pixel on the screen. Writing a 1 into a memory cell causes the display to generate a dot on the screen. Images can be constructed on the screen by writing into the appropriate group of memory cells.
- cdr-coding
  This is a technique used in the hardware for making the storage of lists more compact. In every memory word, two bits are used to indicate whether the word following a preceding word is one of the following:
  - Not part of a list (cdr-nil)
  - The address of the next word in a list (cdr-normal)
  - The address of the second word in a list (cdr-next)

↗ Chaosnet
The Chaosnet is a baseband packet-switched local area network, similar to the Ethernet. It is used for communication between computer systems and peripherals, and their users. It was originally developed at M.I.T., and it was used by Symbolics to interconnect LM-2 computers. It is typically run at a slower rate than the 10-Mbit/sec Ethernet.

↗ Common Lisp
This is a dialect of Lisp developed by a committee of Lisp implementors from 1981 to 1983 for the purpose of defining a standard for portable software. Common Lisp is a lexically scoped, upward-compatible extension of the Maclisp dialect. (Maclisp was developed at M.I.T.) Symbolics Common Lisp is a compatible superset dialect of Common Lisp developed by Symbolics.

↗ Compiler
A compiler translates a source language into a lower-level form, directly executable by the processor hardware. Compilers usually perform several passes over the program text. During these passes, the code is transformed in various ways to make it more efficient.

↗ Compile time
This refers to the time when a program is compiled, which is often different from the time when it is executed or run. (See Run time.)

↗ Data-type checking
On the 3600, data elements are classified as instances of specific types. Data types include integers, floating-point numbers, character strings, flavor instances, and others. The 3600 hardware performs automatic run-time data-type checking as a part of instruction execution. This ensures that the data types match the instructions. In this way, erroneous operations such as "add a number to a character string" are avoided.

↗ Digital-to-analog converter
A digital-to-analog converter (DAC) changes a stream of individual digital numbers or *samples* into a continuous analog voltage signal. For high-fidelity sound, this conversion is carried out at twice the maximum audio frequency desired.

After conversion, the analog signal emerging from the DAC is sent to a filter, amplifier, and loudspeaker to produce sound.

↗ Ethernet

The Ethernet is a baseband packet-switched local area network for communication between computer systems, peripherals, and their users. The physical structure of the Ethernet is that of a coaxial cable connecting all the nodes on the network. The Ethernet is used to interconnect individual 3600 systems and peripherals into a network.

↗ FEP

The FEP (front-end processor) is an MC68000 microprocessor attached to the 3600 processor. The FEP boots the 3600, manages error-logging, runs diagnostics, and handles MULTIBUS input/output.

↗ Flavors

The Flavor System is Symbolics' implementation of the language features that support object-oriented programming. *Flavors* are the abstract types; *methods* are the generic operators. The objects are *flavor instances* that you manipulate by sending *messages*, which are requests for specific instances of an operator. A feature of the Flavor System is that flavors can inherit instance variables and methods from other flavors in a nonhierarchical way.

↗ Frame buffer

The frame buffer or *bit map* is a two-dimensional memory array with one memory cell for each pixel on the screen. On the 3600 color display, each memory call contains up to 32 bits. (See Bit-mapped display.)

↗ Garbage collection

Garbage collection is a set of techniques for recovering parts of the address space that have been used to represent objects that are no longer accessible. Once the memory is recovered, it can be used again to represent new objects.

↗ Generic algorithms

These are algorithms which can work on any object that obeys a particular protocol, no matter how the object performs the algorithm. For example, a generic Print algorithm works with several kinds of objects, such as integers, floating-point numbers, and arrays.

➚ Generic file-system access
This means that the 3600 supports access to several different file systems on different operating systems and computers. They are all accessed via a uniform, *generic* protocol.

➚ Interpreter
An interpreter is a program translator that executes individual lines of source code in a single pass, usually in an interactive mode.

➚ Interrupt
An interrupt is an event detected by the hardware that changes the normal sequence of program execution.

➚ Laser graphics printer
The Symbolics LGP-1 is a laser graphics printer. This means that it uses a computer-controlled laser to record an image on a drum inside the printer. Standard photocopier technology is used to copy the image from the drum onto a piece of paper.

➚ L bus
The L bus is the high-speed data path which interconnects the 3600 processor to the disk controller, the network controller, the console, the FEP, and main memory.

➚ LIL
LIL is the systems programming language for the front-end processor and the 3600 console, which are both based on the MC68000 microprocessor. LIL has a Lisp-like syntax, with many operations that are close to the instruction sets of processors like the MC68000.

➚ LM-2
The LM-2 is another Lisp Machine made by Symbolics. Introduced in 1981, it is the predecessor of the Symbolics 3600.

➚ Macroinstruction
The Lisp compiler translates Lisp source code into a sequence of macroinstructions — machine-language instructions which are directly executed by the machine. These are distinguished from *microinstructions*. (See Microinstruction.)

➚ Message-passing
See the Flavor System.

➚ Microcode
The microcode is the low-level software that controls the

internal data paths within the processor to execute macroinstructions. For example, a multiply macroinstruction causes the processor to execute a sequence of microinstructions which fetch the operands, perform the multiplication, and store the results in the appropriate place.

↗ Microinstruction

A microinstruction controls the internal datapaths of the processor. A microinstruction is stored in a microword. (See Microcode.) On the 3600, a microinstruction is 112 bits wide. Microinstructions require the interval of one clock cycle (microcycle) for their execution.

↗ Mouse

The mouse is a hand-held pointing device attached to the 3600 system. It has three buttons on top of it and a tracker ball underneath it. When you move the mouse across a table top, the cursor on the bit-mapped display screen follows the mouse's movements.

↗ NanoFEP

The NanoFEP is a tiny microcomputer that serves as a front-end processor to the MC68000-based front-end processor (FEP). **The primary tasks of the NanoFEP are bootstrap** loading the FEP and monitoring the physical environment of the processor cabinet.

↗ Object-oriented programming

See the Flavor System.

↗ Packages

The package system is a means of identification for code modules. The main goal of the package system is avoiding conflicts between the names of functions and other Lisp objects written by different people for different purposes. Package names prefixed to Zetalisp functions allow functions with the same name to coexist in a single environment, without ambiguity or conflict.

↗ Page

A page is the unit of memory used for the purposes of virtual memory management. On the 3600, a page is 256 36-bit words.

↗ Pixel

This stands for picture element, a point on a bit-mapped display screen.

↗ Pointer

A pointer is a reference to a word in virtual memory.

↗ Run time

This refers to the time when a program is executed or run, in contrast to compile time. (See Compile time.)

↗ Stack

The stack is an area of memory reserved for temporary storage. A stack adheres to a last-in, first-out protocol. As objects are "pushed" onto the stack, the stack pointer is incremented. As objects are "popped" from the stack, the stack pointer is decremented. In the 3600, special high-speed memories are reserved as stack buffers.

↗ Stack groups

Stack groups are Lisp objects which represent a computation and its internal state, including the Lisp stack. The stack group holds state information, including the local environment, special variables, and dynamic storage associated with a computation.

↗ Streams

Streams are a kind of "software channel" used to implement input and output in Zetalisp. Many streams are implemented with flavors.

↗ Symbolics Common Lisp

This is a compatible superset of the Common Lisp standard, currently under development at Symbolics.

↗ Virtual memory

From the programmer's viewpoint, the virtual memory space is the space in which programs and data are contained. From the system's viewpoint, the virtual memory space is a collection of *pages* of information, some of which reside in physical memory and some of which reside on disk. Since memory can be read or written only when it is resident in physical memory, the 3600 hardware automatically swaps between physical memory and the pages on the *paging disk*.

↗ Winchester disk

A Winchester disk is a mass-storage medium consisting of one

---

or more spinning magnetic platters and a low-mass recording/reproducing head. The disk medium is sealed to keep out dust, and therefore, it is not removable from the drive.

↗ Zetalisp

Zetalisp is a dialect of Lisp used on the 3600. It was originally based on the Maclisp dialect developed at M.I.T., but includes many extensions useful in the Lisp Machine environment.

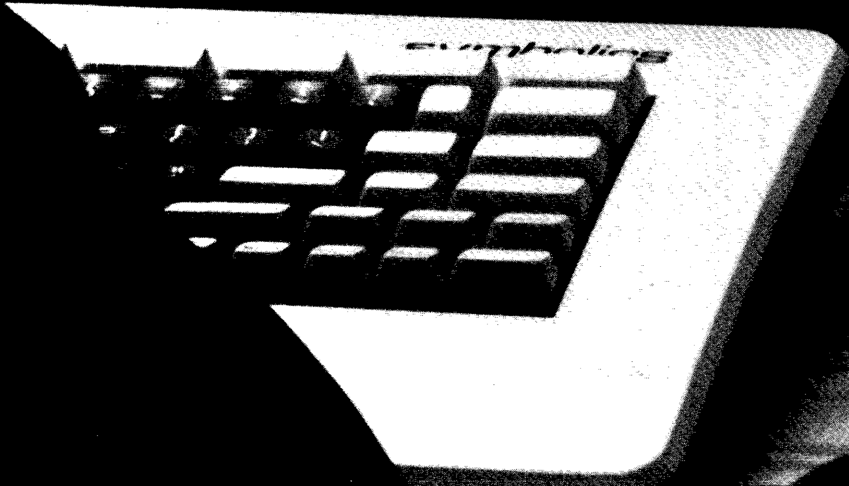
↗ Zmacs

Zmacs is the text and program editor on the 3600. It is a real-time display-oriented editor. This means that the text being edited is always visible, and commands are executed immediately.

↗ Zmail

Zmail is an elaborate interactive system on the 3600 for reading and sending mail.

symbolics



Index

Actual file name 65
Address space 131
Areas 131
Arrays 31
Auto-dial Feature 7, 10, 75

Backup facilities 63
Binding stack 84
Bit-mapped display 131
Bitblt 131
Block mode operations 101
Bootstrap loading 106

Cartridge tape drive 124
Catch/Throw 33
Cdr-coding 81, 131
Chaosnet 132
Character set 34
Color display system 118
Common Lisp 46, 132, 136
Compile time 132
Compiler 132
Console cabinet 113
Control stack 84
Control structures 32
Converse 73

Data stack 84
Data types 82
Data-type checking 132
Deletion, hard 25
Deletion, soft 25
Diagnostics 107
Digital audio output system 109
Digital-to-analog converter 132
Direct memory access 101
Disk controller 110
Display Debugger 21

Ethernet 117, 133

FED 25
FEP 101, 133
FEP file system 108
File system editor 25
Fill mode 121
Filter 71
Flavors 32
Font editor 25
Formatted output 35
FORTRAN 77 15, 49
Frame buffer 133
Frames 84
Front-end processor 105

Garbage collection 133
Generic algorithms 133
Generic file system 134, 64
Generic functions 80
Gray plane 25

Hard fault 97
Hardware error signals 106

IEEE Floating-point 31
Immediate number word format 80
Input/output streams 35
Inspector 21
Instantiation 39
Instruction cache 93
Instruction fetch unit 89
Instruction formats 85
Instruction set 85
Interlisp Compatibility Package 47
Interpreter 134

L bus 100, 134
Lexical scoping 51
LGP-1 laser graphics printer 134, 126
LIL 15, 47, 50, 67, 106, 134
Lisp Machine File System 62
LM-2 4, 134
Logical file name 65

Loop iteration macro 33

Macroinstruction 134
Macros 45
MACSYMA 52
Manchester encoding 110
Manchester-encoding logic 109
Map Cache 97
Marking text 16
Message-passing 134
Message-passing programming 38
Method 32
Microcode 89, 134
Microinstruction 135
Microtasking 89
MMPT 96
Mouse 10, 135
MULTIBUS 67, 105
Multiprocessing 34

NanoFEP 108, 114, 135
Numerical types 31

Object-oriented programming 38

Package system 36, 135
Page 135
Page tag bit 99
Page Tags 98
Paging policy 65
Patch system 29
Pathname 65
Peek 23
PHT 96
PHTC 96, 97
Plane mode 121
Planes 32

Real-time interrupts 106
Real-time process 67, 106
Remote login 75
Run-time data-type checking 79

Scheduler 66
Screen editor 56
Scrolling 61
Serial lines 105
Site file 65
SMPT 97
Sort function 36
Spy bus 107
Stack buffer 84
Stack buffers 93
Stack group 84
Stack groups 34, 136
Status line 58
Storage module drive 110
Streams 34, 35, 64
Structures 32
Supdup 67
Symbol processing 5
Symbolics Auto-dial Feature 10
System 28
System declaration 28

Tagged pointer word format 80
Tape drive 124
Telnet 67

UNIX 64, 65
Unwind/protect 33

VAX/VMS 64
Virtual memory 136
Virtual memory management 65
VPN 97

Winchester disk 136
Window frame 56
Window pane 56
Window system 55

Zetalisp 5, 15, 28, 29, 31, 32, 33, 34, 35, 137
Zmacs 13, 14, 15, 17, 137
Zmail 69, 71, 137

3600 Technical Summary
990098

Design: Schafer/La Casse
Photography: Thayer & DeMaio
Photograph Page 100: Robert Stone
Typesetting: Cover – Litho Composition Co.
Printing: Henry Sawyer Co.