March, 1988
Order Number: 311532-002

iPSC®/2

USER'S GUIDE

(Preliminary)

ADVANCE INFORMATION

This is a draft copy of the manual. Material in this document is for informational purposes only and is subject to change without notice. Intel Corporation assumes no responsibility for any errors which may appear in this document.

intel Corporation

REV.	REVISION HISTORY	DATE
-001	Original issue	12/87
-002	Revision	03/88

RESTRICTED RIGHTS

CHAPTER 1 - The iPSC/2 SYSTEM

CHAPTER 2 - iPSC/2 SYSTEM ARCHITECTURE

CHAPTER 3 - iPSC/2 SOFTWARE ARCHITECTURE

CHAPTER 4 - USING THE iPSC/2 SYSTEM

TABLES

FIGURES

CHAPTER 1

THE iPSC®/2 SYSTEM

INTRODUCTION

This manual is the first in a multiple-volume set describing Intel's Personal SuperComputer, referred to as the ***iPSC®/2***. The iPSC/2 is the second generation hypercube in a family of powerful concurrent processing systems. This manual provides an overview of the iPSC®/2 system, a massively parallel computer system that is scalable in size. Each node in the system is a self-contained computer with substantial processing and memory capabilities. A high-speed network, optimized for message passing, connects these nodes.

PURPOSE

The purpose of this manual is to provide background information necessary to understanding the iPSC system. This manual provides a functional overview of the system's hardware and software and explains the basic concepts needed to understand this concurrent machine.The purpose of this manual is to provide a detailed overview of the iPSC/2 system. It is divided into four chapters:

- **The iPSC/2 System** — A basic description of the system. A table outlines the major differences between the earlier iPSC and the new iPSC/2.
- **System Architecture** — A description of both hardware and software architectures. Provides more detailed information of the topics covered in Chapter 1.
- **Software Architecture** — Details of the software including message passing and I/O.
- **Using the iPSC/2 System** — Concepts you must know in order to run programs on the system.

Detailed system programming information is included in the *iPSC/2 C Programmer's Reference Manual* and the *iPSC/2 FORTRAN Programmer's Reference Manual*.

SYSTEM OVERVIEW

The iPSC/2 can be viewed as an ensemble of processing nodes, each connected to the other via message passing on the network. Solutions to problems are computed by solving a portion of the problem on each of the processing nodes. The system consists of four main functional components:

- The Cube
- The Nodes
- The System Resource Manager
- The Network

The Cube

The iPSC/2 consists of a large number of high-performance processors that each work concurrently on parts of a larger problem. Each processor...along with its associated memory...is referred to as a ***"node"***. The nodes are connected together by high-speed communication channels to form a self-contained ***"cube"*** housed in a free-standing enclosure. You may have 1, 2, or 4 of these enclosures...thereby allowing you to have functionally larger cubes in your system. Based on the number of enclosures, you may have 16, 32, 64, or 128 concurrent processors in your cube. The cube houses the individual nodes (processors) which are connected in a hypercube topology.

The Node

Each iPSC/2 node contains a 32-bit microcomputer with the execution speed and memory capacity of a typical superminicomputer. The node's modular hardware organization and packaging allows it to be optionally configured with extra memory, a fast scalar floating-point unit, or a fast vector floating-point unit.

Each node is also equipped with a Direct-Connect™ routing module for high-speed message passing within the system's hypercube communication network. In addition to the seven communication channels used to connect up to 128 hypercube nodes, each routing module provides an eighth channel for high-speed external communication.

Each node of an iPSC/2 system can vary in performance and capability. The basic node consists of the Intel 80386 processor, an 80387 numeric coprocessor, local memory ranging from 1 to 16 megabytes, and a Direct-Connect module.

A basic processing node can be augmented with two different numeric accelerator options: the VX Vector Processor and the SX Scalar Processor. The VX Vector Processor option was first made available on the iPSC system, and forms the basis for the supercomputer numeric performance in the iPSC family.

The System Resource Manager

As shown in Figure 1-1 below, the system resource manager (SRM) serves as the iPSC/2's connection to the outside world. The SRM has three main purposes:

- Administrative console for the system
- Gateway to other computers and workstations connected by Ethernet to the system
- Host for various development tools

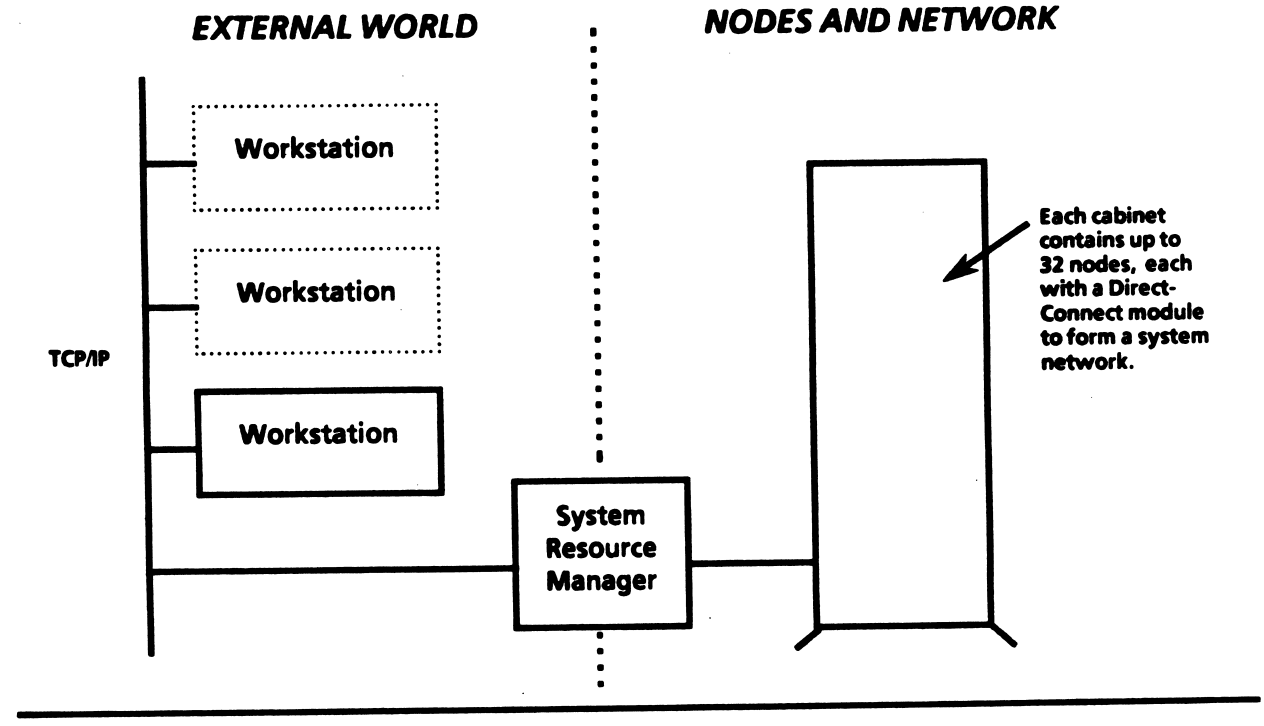

Figure 1-1
iPSC/2 System

As the administrative console, the SRM hosts the UNIX System V.3 operating system and contains a Direct-Connect communications module for accessing the iPSC/2 system. Thus, the SRM has the same communications performance when accessing the cube as do the nodes within the system.

The SRM also serves as a gateway to other computers and workstations connected to the iPSC/2 system by TCP/IP Ethernet. This makes it possible for users on other workstations to use the software tools resident on the SRM, such as the compilers, debuggers, and diagnostics. This means that you can develop applications from an already familiar environment.

The SRM is the host for various development tools. These Concurrent WorkBench™ tools (described in the following paragraph) can be accessed directly by logging into the SRM.

Concurrent WorkBench

The Concurrent WorkBench is a software package that transforms a developer's workstation (such as a Sun-3) into a workbench for developing and running concurrent applications on the iPSC/2 system.

With this software, you not only have access to and control of iPSC/2 system resources, but also you retain all of the windowing, networking, file system, and other features available on the workstation. Multiple workbenches can access the same iPSC/2 system simultaneously.

The Concurrent WorkBench tool collection includes the following:

- C Compiler
- FORTRAN Compiler
- Common LISP Environment
- DECON Concurrent Debugger
- FORTRAN Vectorizer
- Vector Math Library
- Vector Microcode Tools

As shown in Figure 1-2 below, the Concurrent WorkBench extends the programming environment of the iPSC/2 system to independent workstations by using an Ethernet (TCP/IP) connection. It also brings the message-passing facilities out of the iPSC/2 system. Thus, programs resident on these workstations can use the same message-passing constructs as programs on the iPSC/2 node. This is accomplished by simply linking the program with the appropriate Intel-supplied library.

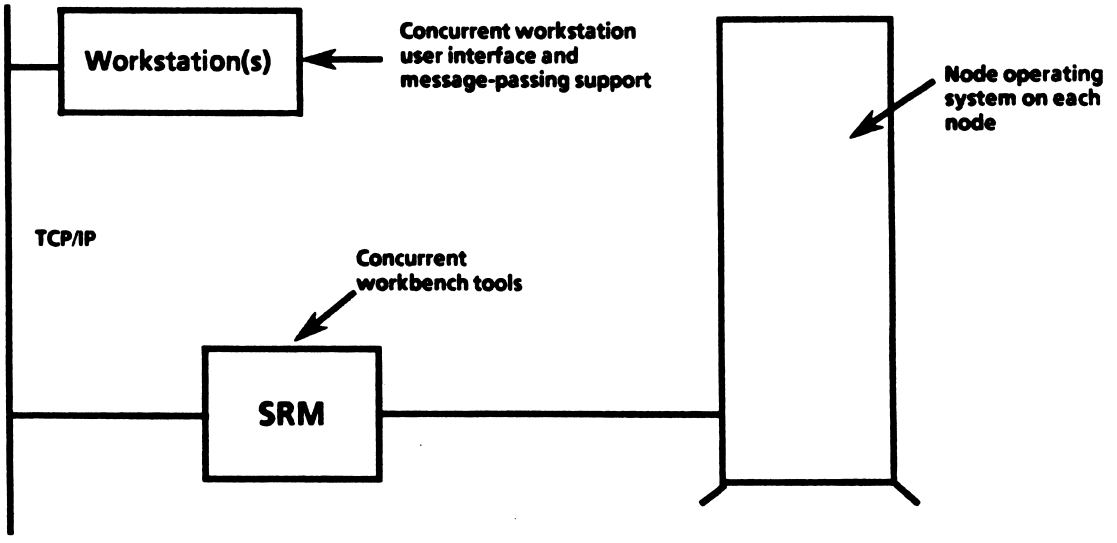

Figure 1-2
iPSC/2 Concurrent WorkBench Interface

The Network

Although physically implemented as a hypercube, the Direct-Connect™ communications hardware and operating system software allow you to assume (and achieve) communications as if every node of the system were directly connected to every other.

The Direct-Connect hardware can be thought of as a switching network. When one node wants to communicate with another node, a series of switches are closed and the communication path is established. Once this communication path is built, messages are passed at the full hardware speed of 2.8 Mbytes/second. The only time a node processor is involved with a message is at the source or the final destination of a message. The other processors in the path continue with their normal activities. Because it takes only a few microseconds per node to build this path, both single-hop and multi-hop message transmittal is uniformly fast.

The NX/2 operating system manages message flow control and minimizes deadlock through message passing protocols and controlled buffering. Once sufficient buffer space is available in the destination node, the entire message is transmitted at once.

Programs exchange messages with other programs using a standard interface supported by the system, common to all languages in the iPSC/2 system.

PACKAGING

The iPSC/2 is a modular system that allows easy upgrading. In addition to the basic cabinet unit and nodes, other modules include numeric accelerators, memory modules, Direct-Connect™ communication module, and software modules.

Cabinet Unit

iPSC/2 nodes fit into modular cabinet units. Each cabinet unit:

- provides 32 slots for node boards or expansion boards
- contains two additional slots for a Unit Service Module (USM) and a spare node board

Up to four cabinet units can be configured as one system, with up to 128 basic nodes or 64 nodes with expansion boards. The USM provides initialization and diagnostic services for each cabinet unit.

A panel on the front door of each cabinet unit displays the red and green status LED indicators of each board within. Cabinet units are mounted on sturdy casters, and can be moved about a floor easily. The cabinet units are secured in place with stability arms splayed outward at each corner.

The iPSC/2 system has only ordinary power and cooling requirements, needing 220 volts AC phased power at approximately 20 amperes per unit. European specification systems are available.

Node Modules

- The Direct-Connect Module — This module is mounted on each iPSC/2 node board and can be easily replaced or updated in the field. Future enhancements in communication technology can be added without replacement of entire nodes.
- Numeric Accelerator Modules — An arithmetic module can be added to any iPSC/2 node. The module can be the standard 80387 arithmetic co-processor or a scalar arithmetic accelerator module. The iLBX-2 interconnect is maintained for the iPSC-VX vector accelerator board, and for future enhancements.
- Memory Modules — Additional memory boards are available for expanded memory requirements.
- Software Modules — Modularity in the NX/2 operating system allows the addition of new features and the dedication of some portions to hardware.

System Resource Manager

The System Resource Manager is housed in a separate enclosure which also contains 140 Mbytes of hard disk storage, a cartridge tape drive, a Direct-Connect module, and a TCP/IP Ethernet connection.

NEW FEATURES

If you are a current iPSC/1 user, it is helpful to know the major differences between the iPSC/1 and the iPSC/2 system. The following table highlights these differences.

Table 1-1
iPSC/1 vs iPSC/2

Topic	iPSC/1	iPSC/2
Node Processor	80286/80287	80386/80387
Message Passing	Messages are routed from node processor to node processor to reach the destination node processor via LAN chips	Messages can be sent directly from any node processor to any other node processor in the system via the direct-connect module
Remote Workstations	None	Multiple remote workstations can share the system and the software development tools on the system.
Numeric Accelerators	VX Vector Processor	VX Vector Processor SX Scalar Processor
Cube Sharing	None	Multiple users can share the cube.
Node Memory Options	4 Megabytes/board	Memory available in: 1 MByte/board 4 Mbytes/board 8 Mbytes/board 16 Mbytes/board
CubeManager/SRM	80286 based Multibus I XENIX operating system	80386 based AT bus UNIX V.3
Host File System	Limited	Host file system has access to node programs. Standard C and FORTRAN calls can be made from node programs
Network Access	Optional Excelan	Integrated Excelan

APPLICABLE DOCUMENTS

The iPSC/2 Documentation Set is composed of both iPSC/2 and UNIX manuals. The basic iPSC/2 manuals are listed in Table 1-2, applicable UNIX documents are listed in Table 1-3, and manuals for options are listed in Table 1-4.

Table 1-2
iPSC/2 Applicable Documents

Title	Order Number	Description
User's Guide	311532-002	Provides an overview of the iPSC/2 system including both hardware and software architecture.
C Programmer's Reference Manual	311017-002	Provides detailed information for all C routines and commands for the iPSC/2
FORTRAN Programmer's Reference Manual	311019-002	Provides detailed information for all FORTRAN routines and commands for the iPSC/2
System Administrator's Guide	311014-002	To provide a detailed description of the system administration tasks related specifically to operating and maintaining the iPSC/2 system.
iPSC/2 Simulator Manual	311534-002	The simulator is a software program that simulates the actions of the Intel iPSC/1 and iPSC/2. It allows development of iPSC programs in a controlled environment prior to execution on the actual cube.
iPSC/2 Concurrent Debugger	311569-001	Describes the DECON debugger which is a source level debugging tool for C and FORTRAN applications on the iPSC/2 system.
Green Hills FORTRAN Language Reference Manual	311020-002	Describes the FORTRAN compiler for the 80386. It explains FORTRAN-386 language features and extensions. It also describes the calling conventions, register and memory allocation strategies, program optimization tools, and methods of porting programs to FORTRAN-386.
Green Hills C Language Reference Manual	311567-001	Describes the C compiler for the 80386.
iPSC/2 Installation Guide for Double-Unit System	311461-001	Describes installation and powering up the iPSC/2 system.
SYP301 Installation & User's Guide		Describes installation and startup for the system resource manager. Also provides hardware technical data.

NOTE The Appendix of this manual includes a special UNIX document not found elsewhere in the manual set. This document is *"An Introduction to the C Shell"* by William Joy. It also contains four appendices including a glossary of terms.

Table 1-3
UNIX Applicable Documents

Title	Description
UNIX System V Administrator's Guide	Describes system maintenance tasks performed on the system resource manager under UNIX.
UNIX System V Administrator's Reference Manual	Describes the UNIX system commands used by system administrators.
UNIX System V User's Guide	Provides a general description of UNIX.
UNIX System V User's Reference Manual	Describes all the UNIX system user commands.
UNIX System V Programmer's Guide	Describes the UNIX system programming environment, and provides detailed descriptions of programming tools.
UNIX System V Programmer's Reference Manual	Contains descriptions of commands, system calls, subroutines, libraries, file formats, macro packages, and character set tables.
UNIX Network Programmer's Guide	Describes the UNIX Transport Interface.
UNIX Streams Primer	Describes Streams, which is a support for networking services.
UNIX Streams Programmer's Guide	Explains how to use the Streams mechanism at user and kernel levels.
UNIX Utilities Release Notes	Contains informaion on UNIX release 3.0
UNIX Quick Reference Guide	UNIX commands, buzzwords, C shell hints, and standard directory layout.
An Introduction to the C Shell	Includes a manual on the C Shell and four appendices: CSH, Less Program, Strings, and a Glossary. ***This document is in the Appendix of this manual.***

Table 1-4
iPSC/2 Manuals for VX Option

Title	Order Number	Description
iPSC/2-VX User's Guide	311570-001	Provides information for developing programs for the iPSC/2-VX (vector processor system). Gives an overview of hardware and software, outlines program development steps, and describes utility routines for the vector processor.
VAST-2 User's Guide	311571-001	Describes VAST-2 which is a pre-compiler that generates code for the iPSC/2-VX vector processor. It vectorizes DO and IF loops in FORTRAN programs.

NOTE The *VecLib Library* is a FORTRAN-callable math function library that complements the vectorizing capability of VAST-2 by allowing explicit representation of vector operations in the program.

Although VecLib is summarized in the *iPSC-VX User's Guide*, detailed information on VecLib is contained in the *FORTRAN Programmer's Reference Manual*.

SPECIFICATIONS

iPSC®/2 SYSTEM	d3	d4	d5	d6	d7
Number of Nodes	8	16	32	64	128
Aggregate Memory (MBytes)					
Basic & SX System					
1 MByte/node	-	16	32	64	128
4 MByte/node	-	64	128	256	512
8 MByte/node	-	128	256	512	1024
16 MByte/node	-	256	512	1024	-
VX System					
1 MByte/node and 1 MByte VX	16	32	64	128	-
4 MByte/node and 1 MByte VX	40	80	160	320	-
8 MByte/node and 1 MByte VX	72	144	288	576	-

iPSC/2 NODE

Node Processor	Intel 80386, native mode execution
Numeric Coprocessor	Intel 80387, arithmetic coprocessor 32-, 64-, 80-bit floating-point (IEEE 754)
Memory	1, 4, 8, 16 MByte modules, 64 KByte zero-wait-state cache
Communication	Direct-Connect™ Routing, implements a fully-connected network, variable length messages support peak data rates of 2.8 MBytes/second on 8 bi-directional connections
Expansion Port	iLBX-II interface to adjacent slot
Indicators	Red, green, and amber
Operating System	NX/2, the Node eXecutive providing process management and message passing
Size	Eurocard 2x4 (9.2"x11")

SYSTEM RESOURCE MANAGER

Central Processing Unit	Intel 80386, native mode execution
Numeric Processing Unit	Intel 80387, arithmetic coprocessor 32-, 64-, 80-bit floating-point (IEEE 754)
Memory	8 MBytes
Cube Communication	Direct-Connect™ Routing, 2.8 MBytes/sec. bandwidth
External Communication	Ethernet TCP/IP local area network port
Peripherals	140 MByte hard disk 1.2 MByte 5¼" floppy ¼" cartridge tape
Operating System	AT&T UNIX, Version V, Release 3.0

ELECTRICAL & ENVIRONMENTAL

	Cabinet Unit	SRM	Monitor/Keyboard
Electrical			
AC Voltage	230 VAC ± 15%	115/230 VAC ± 10%	115/230 VAC ± 10%
AC Current	16 amps	5/3 amps	0.5/0.25 amps
Frequency	50/60 Hz ± 5%	50/60 Hz ± 5%	50/60 Hz ± 5%
Power	3366 watts	314 watts	50 watts
Safety/RFI/EMI (designed to meet)	UL 478 CSA C22.2 No. 154 VDE 0806 VDE 0871 IEC 380 FCC 15 CFRJ Class A	UL 478 CSA 22.2 VDE 0871 IEC 435 FCC 15 CFRJ Class A	UL IEC 435 FCC 15 CFRJ Class B
Environmental			
Temperature	10-35° C	17-32° C	5-40° C
Humdity	85%, maximum non-condensing	5-85%	10-90%
Altitude	0-8,000 ft	0-7,000 ft	0-8,000 ft
Acoustical	50 dBA, maximum		
Physical			
Dimensions	16"x16"x49"	22"x18"x6"	14"x15"x14" (monitor) 2"x18"x8" (keyboard)
Weight	215 lbs	50 lbs	21 lbs (monitor) 4 lbs (keyboard)

CHAPTER 2

iPSC®/2 SYSTEM ARCHITECTURE

INTRODUCTION

This chapter provides detailed information on the iPSC/2 System Architecture and elaborates on the topics covered in Chapter 1.

PURPOSE

This chapter provides a description of the iPSC/2 system architecture. It is divided into the following parts:

• System	Provides an overview of the iPSC/2 system. This chapter describes the cube, system configurations, and the network.
• Node Processor	Describes the iPSC/2 node processor which is the heart of the system. Each node is an independent, single board, 32-bit microcomputer with its own memory. A brief description of the major node components is included.
• Direct-Connect Modules	Describes the communication network that connects the nodes and provides uniform message latency and high throughput.
• Vector Processor	Describes the optional VX node that provides vector processing capabilities.
• System Resource Manager	Describes the computer that serves as a gateway to other system elements as well as a system administration console.
• Concurrent WorkBench	Describes the software package that converts a workstation into a workbench for developing and running concurrent applications on the iPSC/2.

THE iPSC/2 SYSTEM

As described in Chapter 1, the iPSC/2 system basically consists of four parts: a cube, processing nodes, a system resource manager, and a network.

In order to understand the hardware architecture of the system, it is helpful to look at the packaging which consists of a:

- **The Cabinet Unit** — A housing for the ensemble of processing nodes that make up the system and for the communications module that forms the network. All of the nodes in a system are collectively referred to as a *"cube."*
- **The System Resource Manager (SRM)** — The computer (with monitor and keyboard) that serves as a gateway to other computers and workstations and also serves as the system administration console.

These components are shown functionally in Figure 2-1 below. Figure 2-2 shows the physical components of a single cabinet unit system.

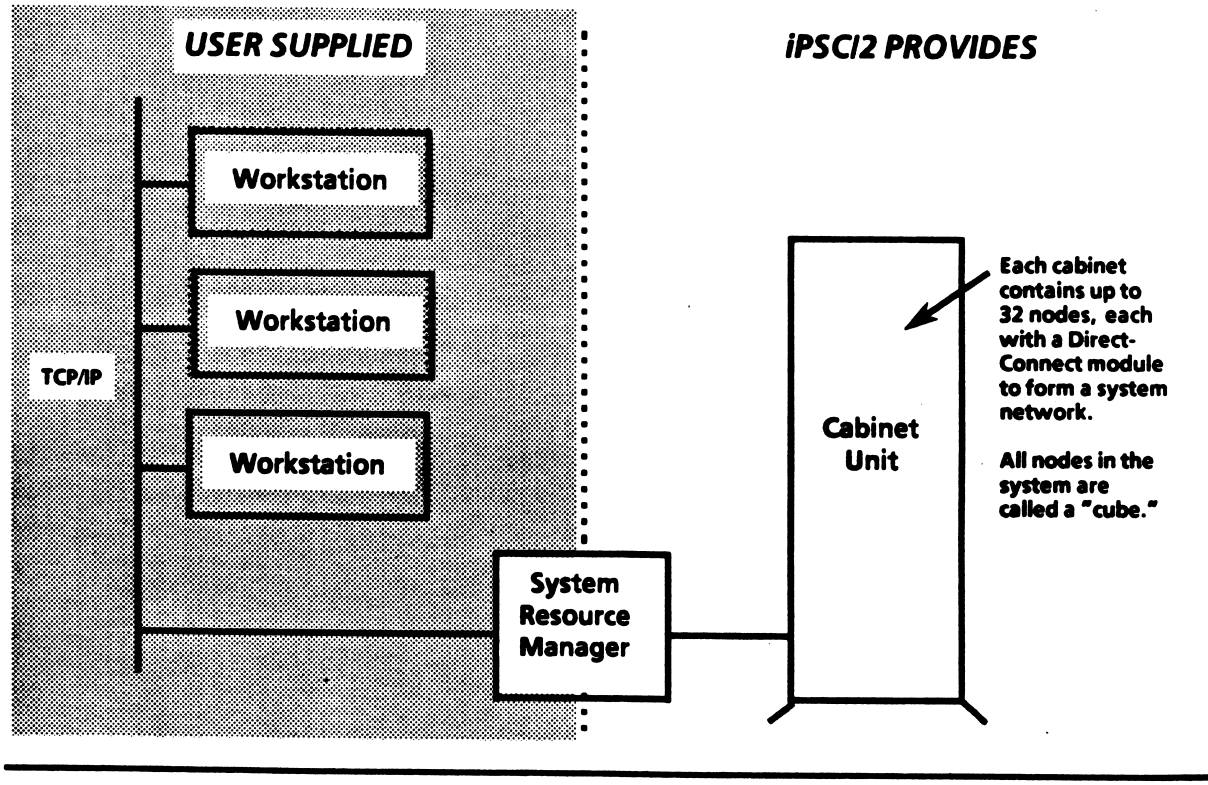

Figure 2-1
iPSC/2 System Packaging Components

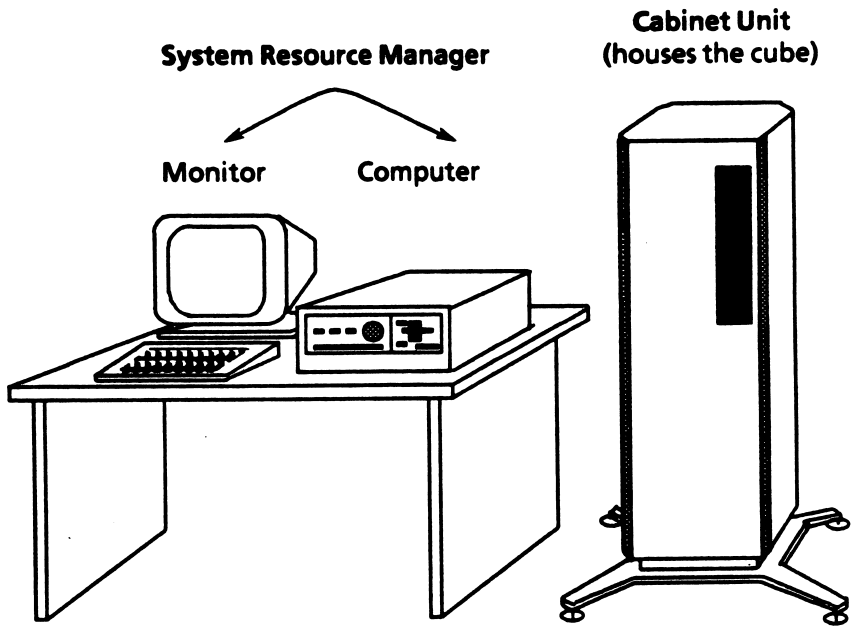

Figure 2-2
Basic iPSC/2 System
(Physical Components)

The Cube

The iPSC/2 is an ensemble of processing nodes, each connected to the other via message passing on the network. Although these processing nodes are physically implemented as a hypercube (see description below), the Direct-Connect™ communications hardware allows other types of topologies (meshes, rings, trees, etc.) to be mapped on to the system. The processing nodes are housed in *"cabinet units."* The iPSC/2 system includes 1, 2, or 4 of these units. All of the nodes in the system are referred to as a "cube."

The cube provides the multiprocessor computational power of the iPSC/2 system. The iPSC/2 can be configured in versions ranging from 16 to 128 processor nodes with memory ranging from 1 MByte to 16 MByte per node for a maximum of 1 GByte per system or cube. A vector concurrent system (iPSC/2-VX) uses a vector accelerator board with each processor in configurations of up to 64 nodes to achieve floating point performance of more than 400 Mflops.

Software commands or routines let you "partition" this cube into one or more (up to 10) subsets of nodes. Each of these subsets is also called a "sub-cube." (Note that this subset can be all the nodes of the hardware cube.) Only one user at a time can use any sub-cube, but several users can use different sub-cubes in the same hardware cube.

The hypercube is a versatile concurrent architecture. This multidimensional structure makes the hypercube well suited to both homogeneous and heterogeneous computational problems. Its communication properties of high data bandwidth and low message latency provide the computational efficiency needed to ensure high performance. Multiple, high-performance 80386 microcomputers work simultaneously, but independently, on parts of a larger problem.

The iPSC/2 avoids the incremental delay associated with "store and forward" message relay mechanisms used in all first-generation hypercubes by employing *Direct-Connect*™ Routing. This results in minimum delay for short messages and maximum communication bandwidth for long messages. You can view the machine as an ensemble of processors with an arbitrary interconnect.

The vector concurrent system provides a dual approach to achieve maximum performance. High-level parallelism uses multiple processors operating concurrently to speed up performance. Low-level parallelism is provided by a pipelined arithmetic accelerator at each node to support high-performance computations for vector and matrix operations.

The Hypercube

A *hypercube* (or *binary n-cube*) is an *n*-dimensional cube where *n* represents the number of directly-connected nodes that establish the cubes dimension.

A hypercube has 2^d identical nodes in which d represents the dimension of the hypercube. Note that the number of nodes is a power of two. Thus, a 2^5 hypercube has 32 nodes and a dimension of 5. The hypercube interconnection for a D5 cube is physically implemented via the backplane within each computational unit. Internal cables connect units in larger dimension cubes.

Because of the versatility of the architecture, other topologies ...such as rings, trees, etc...can be mapped onto the system. These topologies can be realized by using certain techniques in the application code.

System Configurations

The iPSC/2 family of computers includes several different configurations. The basic configurations are:

- **iPSC/2 Standard Systems** — A system can have from 8 to 128 processing nodes with 1, 4, 8, or 16 megabytes of memory per node.
- **iPSC/2 Scalar Systems** — These systems include the Weitek 1167 scalar extension (SX) which is a fast floating-point unit added to the processing node.
- **iPSC/2 Vector Systems** — These systems include the vector extension (VX). Each node processor board has a companion vector processor board.

Table 2-1
iPSC/2 System Configurations

System Type	Model Number	Number of Cabinet Units	Number of Nodes
iPSC/2	D3	1	8
	D4	1	16
	D5	1	32
	D6	2	64
	D7	4	128
iPSC/2 with Scalar Processors	SX/D4	1	16
	SX/D5	1	32
	SX/D6	2	64
	SX/D7	4	128
iPSC/2 with Vector Processors	VX/D4	1	16/16*
	VX/D5	2	32/32*
	VX/D6	4	64/64*

* Each model has an equal number of Node Processor Boards and Vector Processor Boards.

In addition to the various combinations of nodes available for a system, each node can have 1, 4, 8, or 16 megabytes of memory. The aggregate memory available for each iPSC/2 system is given in Table 2-2 below.

In vector extension (VX) systems, the vector processor board uses a board slot adjacent to the node processor board. Therefore, these systems are available with 1, 4, or 8 megabytes of memory but not 16 megabytes. However, there is an additional megabyte of memory on the vector processor board.

Table 2-2
iPSC/2 Memory Sizes
(in megabytes)

System Type	D3	D4	D5	D6	D7
iPSC/2 and iPSC/2 SX					
1 MByte/node	8	16	32	64	128
4 MByte/node	32	64	128	256	512
8 MByte/node	64	128	256	512	1024
16 MByte/node	128	256	512	1024	-
iPSC/2 VX					
1 MByte/node and 1 Mbyte VX	16	32	64	128	-
4 MByte/node and 1 Mbyte VX	40	80	160	320	-
8 MByte/node and 1 Mbyte VX	72	144	288	576	-

The Network

The Direct-Connect™ communications hardware and operating system software provide the communications network among the nodes and between the nodes and other system elements.

Figure 2-3 illustrates the connections between the system resource manager and the cabinet unit, as well as the connections within the cabinet unit.

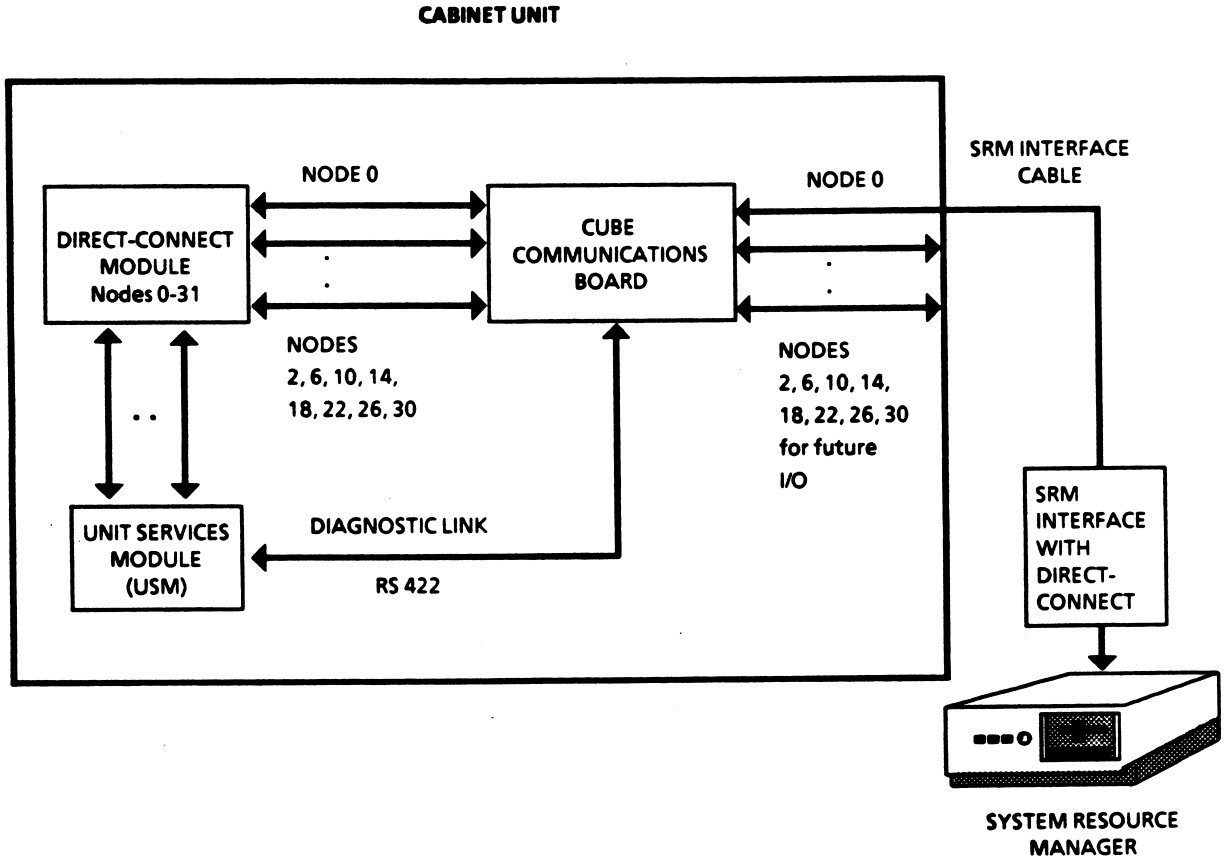

Figure 2-3
Network Connections

The following is a brief description of the main components shown in Figure 2-3. More detailed information on the node processer and system resource manager is given later in this chapter.

- **Direct-Connect**

  All of the nodes are connected to one another by means of custom-designed, Direct-Connect™ communications hardware. This communications network lets you assume that every node of the system is directly connected to every other node. The only time a node is involved with a message is at the source or the final destination of a message.

  More detailed information on the Direct-Connect Module is given in the next section which covers the *iPSC/2 Node Processor*.

- **Diagnostic Link**

  The diagnostic link is a separate path for communicating with the nodes. It links the system resource manager to the unit services module (USM) through an RS 422 port from the cube communications board.

- **Communication Board**

  This board translates node processor 0's eighth channel into signals that can be sent over a cable. (Nodes 2, 6, 10, etc. also feed into the cube communications board, but are not currently used.)

- **Unit Service Module (USM)**

  The USM board can transmit data to all of the nodes and can reset all the nodes. It acts as a low-speed secondary line into the nodes and serves primarily as an initialization and diagnostic channel.

  The USM demultiplexes the diagnostic channel into each node via the monitor bus, which consists of communication, interrupt, and reset lines. In the event of a failure, this alternate path is used to determine if the fault is within a node or within the communication link to the nodes. It is used to reset the nodes, initiate on-board diagnostics, and monitor the results.

- **SRM Interface Cable**

  The SRM Interface Cable handles both data transmission from node 0 and diagnostics from the RS422 link.

- **Cabinet Unit**

Each cabinet unit is a free-standing enclosure which is 49 inches high and 16 inches square with a 26.75" x 26.75" footprint. Each enclosure has 32 board slots, one spare board slot and the USM board in one 34-slot card cage. Communication transceivers, multiplexer, power supplies, air cooling system, and associated cables are also housed in the enclosure.

Additional cabinet units are internally cabled together and node 0 in the first unit (unit 0) handles all communication between the system and the SRM.

A removable key, found in the switch on the front panel (inside the front door), turns on the cooling fans and + 5-volt power. Front panel lights indicate AC on and DC on in that sequence. A power module at the base of the unit houses a filter and circuit breaker which can be externally reset from the rear.

Unit 0 has the only key switch. Thus, in a multi-unit system, unit 0 enables the power-up sequence for all other units.

THE iPSC/2 NODE PROCESSOR

Each iPSC/2 node processor is an independent, single-board, 32-bit microcomputer. The basic node consists of the Intel 80386 processor, an 80387 numeric coprocessor, local memory ranging from 1 to 16 megabytes, and a Direct-Connect™ module. Each node processor has red, green, and amber LED indicators which are used for diagnostic and system purposes.

The node processor's modular hardware organization and packaging allows it to be optionally configured with extra memory, a fast scalar floating-point unit, or a fast vector floating-point unit.

A basic node processor can be augmented with two different numeric accelerator options: the VX Vector Processor and the SX Scalar Processor. The VX Vector Processor option was first made available on the iPSC system, and forms the basis for the supercomputer numeric performance in the iPSC family. The Scalar Processor is a fast floating-point unit that provides increased performance primarily for scalar arithmetic.

Each node processor is also equipped with a Direct-Connect routing module for high-speed message passing within the system's hypercube communication network. In addition to the seven communication channels used to connect up to 128 nodes, each routing module provides an eighth channel for high-speed external communication.

The node also carries its own multi-tasking operating system, called NX/2 (Node eXecutive) which provides reliable and optimal message communications via the Direct-Connect routing system. This operating system is resident on each node in the system.

Each node processor has red, green, and amber LED indicators which are used for diagnostic and system purposes.

Figure 2-4 shows the main components of each iPSC/2 node processor. These components are described in subsequent paragraphs.

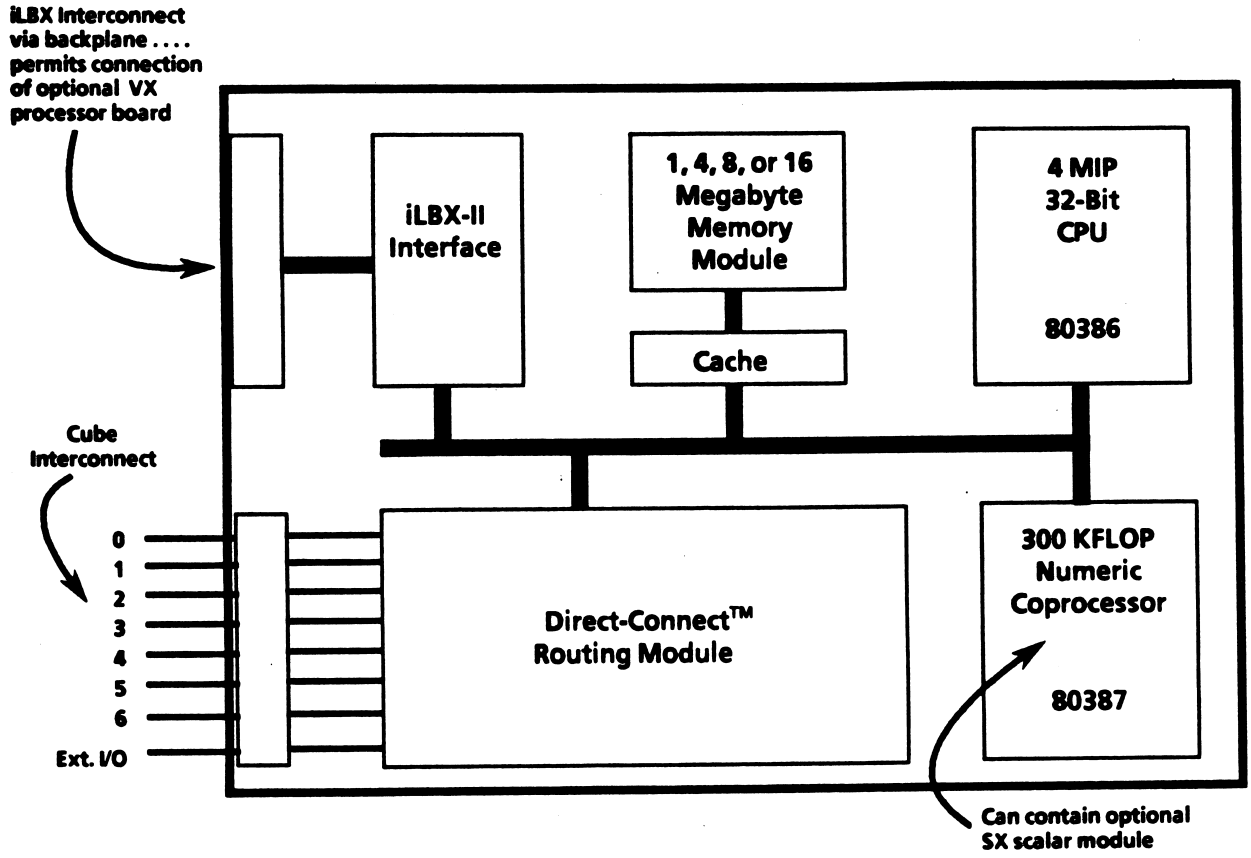

Figure 2-3
iPSC/2 Node Processor

Processor

The heart of an iPSC/2 node processor is the Intel 80386 32-bit microprocessor. Used only in its native 32-bit mode, the 80386 executes FORTRAN, C, and LISP programs at up to 4 MIPS and addresses of up to 16 megabytes of physical memory.

Numeric Coprocessor

Augmenting the 80386 processor is its companion 80387 numeric coprocessor. Also operating at 16 megahertz, the 80387 provides IEEE-compatible scalar floating-point performance of up to 300 kiloflops at 64-bit precision. The 80387 instruction set also includes a number of transcendental and elementary arithmetic functions such as sine, arc tangent, and logarithm.

Fast Floating-Point Unit (Option)

Where an application requires even greater scalar floating-point performance, a node can be configured with the SX Scalar Processor (Weitek 1167). This optional hardware module augments the 80387's capability (the 80387 is retained) and approximately triples the node's peak scalar performance. Directives to the compilers and the linker cause the appropriate SX code to be generated and the proper SX libraries to be linked.

Memory and Cache

Memory accesses from the node processor and floating-point units are buffered by a 64-Kbyte fast static cache memory. The cache is direct mapped and achieves a hit rate of 0.91 for typical applications.

Node memory is modular and may be configured to be 1, 4, 8, or 16 megabytes. In those applications that require very large node memories, two 8-megabyte memory modules may be stacked on a single node. The height of this 16-megabyte module requires that the adjacent slot in the cabinet be left empty.

ROM

The node contains 64 Kbytes of EPROM memory which contains the node confidence tests and the boot loader. There are automatically initialized during system startup.

Direct Memory Access (DMA) Controller

For high-speed transfer of large blocks of data within a node and between nodes via the hypercube network, each node contains a multichannel direct memory access (DMA) controller. One channel is dedicated to inbound messages and another to outbound. Two other channels are available to software, with a total DMA bandwidth of 10.7 megabytes per second.

When performing a DMA, selection is made by the appropriate node Direct-Connect module and then a DMA is performed between memory and the selected node processor.

Communications Network

Each node is equipped with the proprietary Direct-Connect routing module for high-speed node-to-node communication within the hypercube and to the external world. Each routing module supports eight full-duplex serial communication channels operating at 2.8 megabytes per second in each direction and is capable of simultaneously routing up to eight messages. One kilobyte first-in, first-out (FIFO) buffers smooth the transmission and reception of messages between the router and the node, and allow simultaneous transmission and reception.

The ensemble of routing modules that create the hypercube network are capable of routing and relaying messages throughout the system without the assistance of the node processors. The only time a node processor is involved with a message is at the source or the final destination of that message.

Local Bus Interface

To open it to further functional extensions, each node is equipped with a standard, high-speed iLBX™ interface. The iLBX II interface reaches the backplane via one of the two 96-pin DIN connectors. On the backplane, the iLBX bus is routed from each even-numbered board slot to the adjacent odd-numbered board slot. The odd-numbered slots may be populated by boards that extend the memory or processing capacity of the nodes as in iPSC/2 expanded memory systems and Vector Processor systems. Because this is the same interface as used in the original iPSC system, it is compatible with all previous expansion board designs. Transfers across the local bus interface average 10 megabytes per second.

LED Indicators

Each node is equipped with LEDs (light emitting diodes) facing the front door. These indicators are used for simple observation of node status, and for various diagnostic purposes. Three LED's are used, with the green LED under programmer control. These LEDs indicate CPU computation (green) and Direct-Connect activity (red) by default. A third LED (amber) continually indicates numeric coprocessor computation.

NX automatically runs the node board LED's according to the activity in the node. The LED's indicate the current state of the node as follows:

Table 2-3
LED Interpretation

Green LED	Red LED	Amber LED	Meaning
ON, soft pulsing	ON, soft pulsing	OFF	Node operating system is idle, no user processes are loaded.
ON	OFF	OFF	A process is running.
ON	OFF	ON	Math coprocessor is running.
ON	ON	OFF	A process is communicating.
ON	ON	ON	A process is communicating and the coprocessor is running.
OFF	OFF	OFF	All processes are idle. If processes have terminated, use WAITCUBE or KILLCUBE to return LED's to soft pulsing state.
OFF	Flickering	OFF	Fatal error. Bad node hardware or fatal node operating system error. Try BOOTCUBE or CDP.

A process may take explicit control of the green LED by using the LED routine. Once a process has taken control, NX/2 will not run the green LED again until all processes have exited.

Node Operating System

Each iPSC/2 node processor runs its own multi-tasking operating system, called NX/2 (Node eXecutive/2). The NX/2 system provides all of the system services such as: memory management, multiple process control, message passing services, and intertask protection. It was designed specifically to support multiple languages, add as little overhead as possible to executing programs, and ensure reliable, optimal message communications via the Direct-Connect routing system.

Highlights of this operating system are given below. Details are described in Chapter 3 of this guide.

- **Interprocess Communication** — NX/2 provides the program's interface to the Direct-Connect communication network. This communication interface is consistent whether communicating with other processes in the same node, to remote nodes, or to processes in the SRM. Sending and receiving can be synchronous or asynchronous, and individual messages can be typed. All iPSC/2 languages are supported via the NX/2 interface.

- **Process Management** — NX/2 supports multi-processing with round-robin scheduling in each iPSC/2 node. The NX/2 loader takes UNIX Common Object File Format files as input.

- **File Input/Output** — NX/2 supports UNIX-style standard input/output at the node, for each of the iPSC/2 languages. Input/output calls are serviced by the devices on the host system.

- **Physical Memory Management** — Memory space and message buffers are provided for node processes. The highly-efficient memory manager uses the paging mechanism of the 80386 processor, but is not directly visible to your applications.

- **Protected Address Space** — NX/2 also uses the 80386 processor to provide each process with a separate memory space. This hardware enforced protection helps you avoid memory and program corruption.

- **Process and Debugging Support** — Specific support for the DECON Concurrent Debugger is provided by NX/2.

DIRECT-CONNECT™ MODULES

Messages between nodes travel on a fast communication network consisting of a Direct-Connect module (DCM) at each node. This network provides uniform message latency, higher throughput, and NX/2 message services which provide minimal delay for short messages and increased bandwidth for long messages.

Description

In a typical hypercube, messages sent from one node processor to another must pass through a number of intervening node processors (unless the node processors are nearest neighbors) before arriving at the destination node processor. Thus, node processors become involved in handling message traffic that has nothing to do with the information they are processing at the time.

In the iPSC/2, each node contains, in addition to the node processor, a Direct-Connect™ module (DCM) that allows a message to be passed directly from any node processor to any other node processor, passing through only the communications modules without having to pass through intermediate node processors. This is done by a logic switching arrangement. To keep the module simple and efficient, it has only eight paths (7 node paths and a global path).

When a message arrives at a DCM, it contains an identifier indicating which node processor the message is intended for. The message may be intended for the node processor associated with that particular DCM, or for a "nearest neighbor" node processor, or for some other node processor.

- **Node processor & DCM in same node** — When a message arrives at the DCM, it is immediately passed to the node processor in that node.

- **Nearest neighbor node** — Each DCM contains direct lines to all of the DCM's of the "nearest neighbor" nodes. In this case, the message is sent along the appropriate line to the DCM of the destination node, which then passes the message to the node processor.

- **Other node** — In this case, the message is sent along the appropriate line to the DCM of the "nearest neighbor" node that is the closest to the destination node. That DCM then routes it to the next "nearest neighbor" node DCM. This process is repeated until the message is given to the destination node processor. In effect, the message "hops" from one DCM to another until it reaches its destination.

  Note that the node processors themselves are not involved in message passing. A message is routed through the various DCM's, which are simple switching networks...until it reaches the destination node processor.

Messages sent to nearest neighbor nodes take only slightly less time than multi-hop messages to more distant nodes. This is because the most time is used to set up a message for the Direct-Connect module. Once this is done, routing a message through other node Direct-Connect modules takes minimal time.

Uniform Message Latency

Direct-Connect routing dramatically alters the programming methodology used for hypercube machines because the iPSC/2 system does not appear as a hypercube to the programmer. Because of the extremely short time needed to set the switching mechanism at each intermediate node in the hypercube (a microsecond or two), messages traversing the largest iPSC/2 system (128 nodes) take only slightly longer than messages between two physically connected nodes. This results in uniform communication throughout the iPSC/2 system because it appears as if every node were directly connected to every other node.

Because of incremental performance cost for communicating beyond nearest neighbors, first generation hypercubes required programmers to organize the computational problem to minimize messages traversing multiple nodes. Direct-Connect routing imposes virtually no added penalty on multiple node communication. Consequently, you can view the machine as an ensemble of fully interconnected processors.

Communication Network Throughput

Each Direct-Connect Module is equipped with eight bidirectional communication wires. Seven of these are used to interconnect with other nodes in the traditional hypercube topology. The eighth is reserved for future expansion such as dedicated communication to special processors and I/O engines.

Each unidirectional channel, 16 in all, is independent from the others. This allows simultaneous message traffic through a Direct-Connect Module. Because of the dedicated logic, this traffic does not disturb the iPSC/2 node processor. It is not necessary for the node processor to assist in handling messages, even when the message is not intended for the node processor.

With the simultaneous capability of the Direct-Connect Module in the iPSC/2 node, iPSC/2 systems offer an enormous increase in total system network throughput. This is essential for many applications with frequent message traffic, particularly those requiring periodic communication among all nodes in the system at the same time.

NX/2 Message Services

Passing messages is the key to using the iPSC/2's concurrent facilities. A message is sent from one process and received by one or more processes. From a programming point of view, sending a message sends a copy of the contents of a buffer from one process to another. Note that the buffer may be of any data type including the C data type "struct", allowing any amount of data to be sent in one variable. (Note that messages of 0 length can also be sent. These may be used as signals to coordinate node programs and have the advantage of very short transmission time.) The receiving process(es) may be on other nodes, on the host, or even to the same node. The buffer is set to its value(s) in the sending process and a message-sending routine (or subroutine) uses this variable. The message-receiving routine places the copy of the sent value(s) into a local buffer. Messages can be sent and received by both host and node processes.

A call to the **synchronous** message-sending or receiving routines waits until the message is sent or received before the routine returns, allowing the program execution to continue. If the message is not sent or received then the program execution is blocked at the point of the call.

A call to the **asynchronous** message-sending or receiving routines returns immediately and does not wait until the message is sent or received. The program continues execution whether or not the message is sent or received. The routine returns a message id identifying the particular message that has been sent. This feature allows you to do calculations in your program while the message is being sent or received.

Applications experience established the need for efficient support of two different communications requirements: minimum delay for short messages, and maximum communication bandwidth for long messages. The NX/2 operating system provides a standard interface to exchange messages between nodes via Direct-Connect Communications and minimizes the possibility of deadlock.

When dealing with short messages between nodes, NX/2 uses a technique called *"windowing"* to make message passing more efficient. Typically, there is an acknowledgement of each received message. However, when sending short messages, *"windowing"* lets you group two or three messages together but requires only one acknowledgement for the entire group. This minimizes overhead because individual message acknowledgments are not required when the messages are transmitted together.

When dealing with long messages, NX/2 determines that there is a receive buffer waiting or sufficient system buffer available at the destination node. It then, via Direct-Connect routing, streams the long message through at the full bandwidth of the communication system, which is 2.8 megabytes per second.

The size of these messages is limited as follows:

- Host 256 Kbytes - messages larger than this should not be sent to the host.
- Node Depends on the amount of memory in the given node, which can be substantial (up to 16 megabytes).

VX VECTOR PROCESSOR (OPTIONAL)

Supercomputer performance is supported on iPSC/2-VX systems by coupling a high-performance vector processor to each node of the system. Each vector processor has a peak performance that exceeds 6 MFLOPS double precision and 18 MFLOPS single precision. The combined performance for a 64-node system exceeds 400 MFLOPS (64-bit) and more than 1200 MFLOPS (32-bit).

VX NODE ARCHITECTURE

The iPSC/2-VX vector processor (VP) board has a synchronous microprogrammed architecture with a 100-nanosecond cycle time. The VP board's three main functional components are a floating-point arithmetic unit, a data unit, and a control unit, as shown in the Figure 2-5 below and described in subsequent paragraphs.

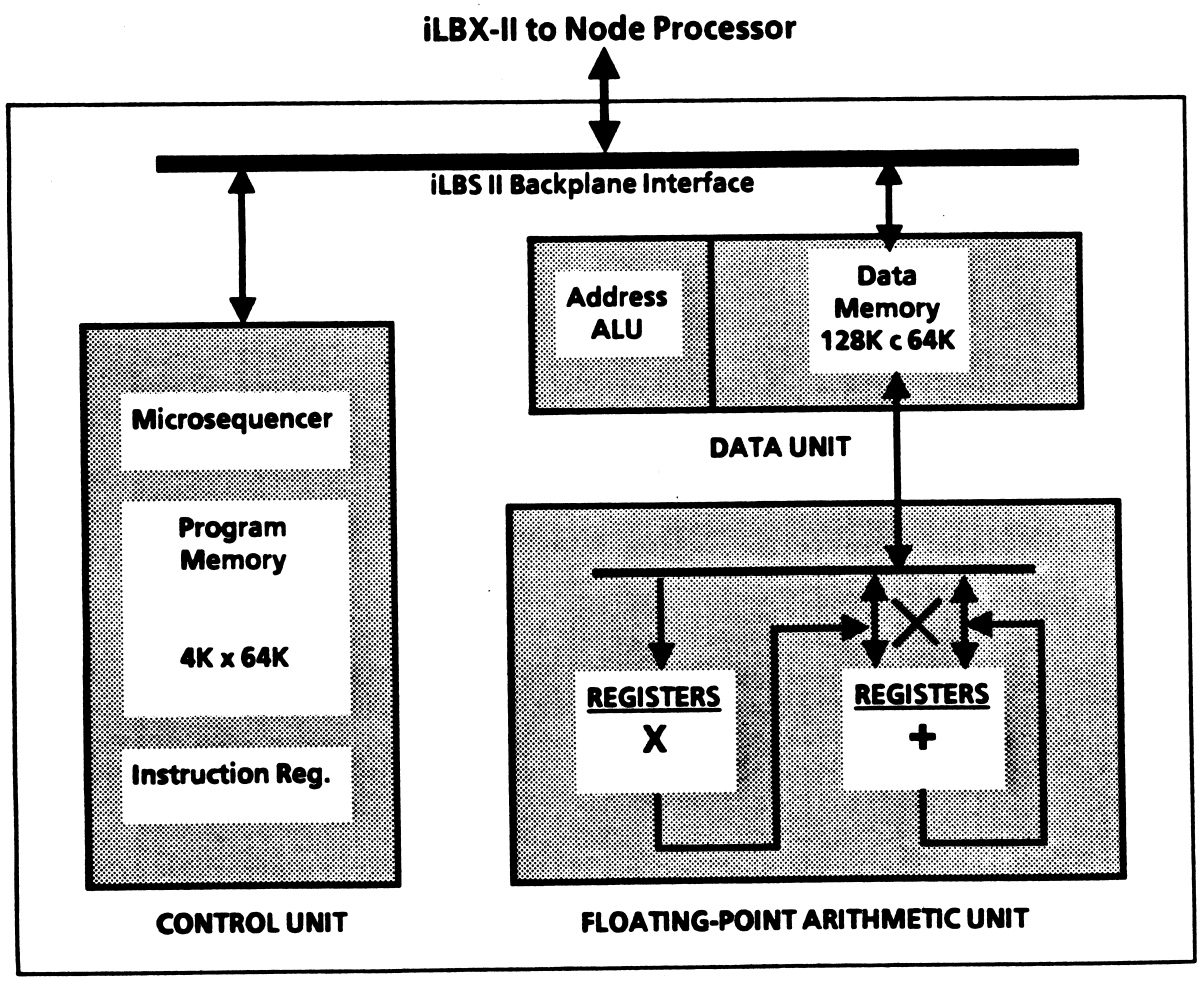

Figure 2-5
iPSC/2 Vector Processor

Arithmetic Unit

The arithmetic unit is a pipelined processor composed of an adder and a multiplier, each with the performance capability shown below. The arithmetic unit also contains 12 registers which are used to store intermediate results and to stage data into the arithmetic elements. Except for gradual underflow, all floating-point operations conform to the IEEE-754 standard. The arithmetic unit also supports 32-bit fixed point and logical operations.

Floating Point Throughput	32-Bit	64-Bit
Adder (elapsed/pipelined)	300/100 ns	300/100 ns
Multiplier	300/100 ns	500/300 ns

Data Unit

The vector processor data unit is composed of a 32-bit address Arithmetic Logic Unit (ALU) and one megabyte data RAM. Data RAM is mapped into the address space of the node processor. This memory space may be used by the 80386 to store user code as well as data. However, computational operands for the vector processor *must* reside in the one megabyte of vector processor data memory.

The vector processor's data memory supports 64-bit accesses in a cycle time of 250 nanoseconds. This memory is augmented by a 16 KByte section of fast (static) RAM, which is capable of 64-bit accesses in 100 nanoseconds. Fast RAM is partitioned into a 4 KByte user-accessible scratch pad and 12 KBytes of high performance storage for intermediate computational results and table functions. The fast scratch pad memory is accessible to the programmer through named FORTRAN COMMON statements. When properly allocated to frequently used scalar or short vector operands, the use of scratch pad memory can significantly improve execution time.

Control Unit

The control unit contains a microsequencer and a 128 KByte program RAM that is directly loadable from the 80386 node processor. Program RAM contains instructions for the microsequencer that allow it to coordinate operation of the vector processor's data and arithmetic units. These microcoded routines include a runtime monitor and an optimized library of vector, scalar, and logical operations.

The vector processor's internal microfunctions may be augmented with additional microcode routines to provide general purpose or proprietary capabilities. The versatility of this arithmetic processor stems from its use of very short pipelines and a 32-bit address ALU which can randomly address data memory of the vector processor.

SYSTEM RESOURCE MANAGER (SRM)

The System Resource Manager (SRM) is a computer with a monitor and keyboard and serves as a gateway to other computers and workstations, as well as a system administration console.

The System Resource Manager controls the cube. It is connected to the workstation by a TCP/IP, Ethernet network link and is connected to the cube by a *Direct-Connect*™ communication link. The SRM provides a UNIX V.3 host for the concurrent workbench, an I/O gateway between the cube and multiple remote workstations, a compile server to the workstations, and management of cube sharing by the workstations.

The System Resource Manager consists of two components: an Intel SYP301 system that combines the power of the 80386 microprocessor with the flexibility of the IBM AT architecture, and monitor. The standard monitor is a Samsung monochrome monitor. The architecture provides a full 32-bit data path for accessing memory. Programs are developed on the System Resource Manager and then loaded into the cube for execution.

The system resource manager, which is a UNIX V.3 based system, contains:

- 16 MHz 80386 CPU
- 16 MHz 80387 math coprocessor
- 8.5 megabytes of 32-bit RAM
- 140 megabyte Winchester Disk
- 5¼-inch floppy disk drive
- ¼-inch cartridge tape drive
- Ethernet network connection
- Direct-connect module

One or more System Resource Managers with attendant cube may be connected in a network to permit accessing the cube from a machine that is not a System Resource Manager. The computer used to control a particular cube is the "host" computer for that cube. The "host" program on the "host" machine will control the "node" programs on the cube.

SYP301 Computer Configuration

The 80386 mother board in the system resource manager provides eight slots which are described in Table 2-4 below:

Table 2-4
System Resource Manager Slots

Slot #	No. of Bits	Contents
1	16	Empty
2	16	Empty
3	16	Excelan board for TCP/IP network capability
4	8	Vega Video7 monitor board
5	8	Tape controller board for cartridge tape drive
6	32	8 megabyte memory board
7	32	301 interface board for communicating with the cube
	16	Disk controller board for 140 megabyte byte Winchester disk

The two empty slots are both 16-bit slots. 8-bit boards can be inserted into these slots as long as the board's software can handle 16-bit interrupts.

THE CONCURRENT WORKBENCH™

The Concurrent Workbench™ is a software package that transforms your workstation, such as a Sun-3, into a workbench for developing and running concurrent applications on the iPSC/2. Now you can access the cube from a workstation just as you can from a terminal connected to the System Resource Manager. You not only have access and control of the iPSC/2 resources but also retain all of the windowing, networking, file system, and other features available on the workstation. Multiple workbenches can access the same iPSC/2 system simultaneously. The Concurrent WorkBench is also used on the System Resource Manager.

The Concurrent WorkBench tool collection includes:

- C Compiler
- FORTRAN Compiler
- Common LISP Environment
- DECON Concurrent Debugger
- FORTRAN Vectorizer
- Vector Math Library
- Vector Microcode Tools

In addition to extending the programming environment of the iPSC/2 system across an Ethernet connection to workstations, the Concurrent Workbench also brings the message-passing facilities out of the iPSC/2 system. Programs resident on supported workstations can use the same message-passing constructs as programs on the iPSC/2 node by linking with the appropriate Intel-supplied library

The various tools on the Concurrent Workbench are described in the following sections.

LANGUAGES

The iPSC/2 system supports FORTRAN, C, and Common LISP languages. The message passing supported by each iPSC/2 language is compatible with the other iPSC/2 languages, and programs of different languages may reside on the same node.

The transition from a 16-bit to a 32-bit node architecture is comparable to the move from minicomputers to super-minis. Thus, the languages and tools selected for the iPSC/2 system are derived from the super-mini and mainframe technology. UNIX V.3 provides the basis for the development environment. The node operating system, NX/2, has itself been completely redesigned to take full advantage of the 32-bit capability of the 80386 node CPU.

Language support comes from production compilers originating in the minicomputer world, which use 32-bit pointers and provides extensive high quality code generation optimizations. For instance, mixed language applications are supported, allowing you to use the language best suited for the different tasks that make up an application: C, FORTRAN, or LISP. FORTRAN not only adheres to FORTRAN-77 standards, but also includes DoD extensions and popular VMS extensions.

C Compiler

Many developers implementing new applications for the iPSC/2 system choose the C language for its flexibility of expression and proximity to the hardware. Most of the tools in the Concurrent Workbench, the node operating system, and the UNIX implementation on the System Resource Manager are written in C. Computer scientists building experimental concurrent software such as operating systems and database systems often choose C for similar reasons.

The C language is a general purpose language which features economy of expression, modern control flow and data structures, and a variety of operators. It is based on the standard Kernighan and Ritchie definition of the C programming language. The C compiler is the Green Hills C compiler.

The standard C language is augmented by a library of pre-defined functions that allow the C programmer to implement the message passing required to build concurrent applications for the iPSC/2. These functions allow C processes to send and receive messages, probe for messages of a given type, and to obtain ancillary information about messages and the iPSC/2 system.

The message passing functions allow different parts of a C application to independently and simultaneously execute on different nodes and synchronize and share information via messages. The so-called asynchronous message passing functions go one step further and allow concurrent applications to overlap message passing between nodes with simultaneous execution of node processes.

C programs that use "logistics" calls are restricted to the host. These calls may call another set of pre-defined functions to manage the logistics of obtaining access to the iPSC/2 nodes, loading the application into the nodes, clearing out the nodes after the application finishes, and flushing any remaining messages in the system. Clearing out the nodes after the application finishes is <u>not</u> restricted and can also be done by the node.

FORTRAN 77 Compiler

Traditional numeric computation and supercomputer applications are implemented in standard FORTRAN. The iPSC/2 FORTRAN conforms to FORTRAN-77 standards, enhanced with VMS-style extensions and the DoD supplemental standard (MIL-STD-1753). Because of the installed base of FORTRAN numeric software, the vector processor enhancement uses FORTRAN-77 calling sequences, as does the automatic vectorizer VAST-2. Most iPSC/2 users implement applications in iPSC/2 FORTRAN.

The FORTRAN language implements the full ANSI FORTRAN 77 standard (ANSI X3.9-1978) with the DoD FORTRAN supplement, MIL-STD-1753. It also implements all of the extensions to FORTRAN 77 documented in the Berkeley 4.2BSD f77 documentation.

The FORTRAN compiler accepts the same compile time options, arguments and libraries as the Berkeley 4.2BSD f77 compiler. The compiler is supplied by Green Hills Software, Inc and is supported by both Green Hills and Intel.

The standard FORTRAN language is augmented by a library of pre-defined routines that allow the FORTRAN programmer to implement the message passing required to build concurrent applications. These routines allow FORTRAN processes to send and receive messages, probe for messages of a given type, and to obtain ancillary information about messages and the iPSC/2 system.

The message passing routines allow different parts of an application to independently and simultaneously execute on different nodes and synchronize and share information via messages. The so-called asynchronous message passing functions go one step further and allow concurrent applications to overlap message passing between nodes with simultaneous execution of node processes.

FORTRAN programs that use "logistics" calls are restricted to the host. These calls may call another set of pre-defined functions to manage the logistics of obtaining access to the iPSC/2 nodes, loading the application into the nodes, clearing out the nodes after the application finishes, and flushing any remaining messages in the system. Clearing out the nodes after the application finishes is <u>not</u> restricted and can also be done by the node.

TOOLS

Tools provided for the iPSC/2 system include those used for the vector processor extension (VX) system, those used with concurrent common LISP, a concurrent debugger, and a simulator.

VX Vector Processor Tools

The iPSC/2-VX system is supported with three types of programming tools: the VX FORTRAN vectorizer, the VecLib math function library, and a microcode development tool kit.

- **VAST-2 Vectorizer** — Provides user access to the power of the VX from a familiar FORTRAN environment. Applications may be ported to the VX system by either vectorizing the code first, performing the concurrent decomposition second, or by the reverse process.

  VAST-2 is a pre-compiler which generates code for the iPSC-VX vector processor . It vectorizes DO and IF loops in FORTRAN programs.

  When VAST-2 processes a FORTRAN program, it creates two files. One is a listing of the input program with diagnostic comments added to tell which loops were vectorized and which were not. If a loop was not vectorized, VAST-2 explains why it was not. VAST-2 also creates an enhanced version of the input FORTRAN program containing vector code in place of the original DO or IF loops

  VAST-2 for the iPSC-VX gives assistance only with the vectorization of tasks on individual nodes, not with the partitioning of the whole program into parallel tasks.

- **VecLib Library** — VecLib is a FORTRAN-callable math function library that complements the vectorizing capability of VAST-2 by allowing explicit representation of vector operations in the program. The VecLib functions include the *Basic Linear Algebra Subprograms* (BLAS) and many other commonly used functions that have been optimized for execution on the vector processor.

- **Microcode Tools** — Certain performance-critical tasks may be implemented as custom microfunctions of the vector processor. The iPSC/2-VX system comes with a complete microcode development tool kit including a microcode assembler and linker to support this optimization technique.

DECON Concurrent Debugger

iPSC applications are composed of multiple processes executing concurrently on up to 128 nodes and exchanging messages with each other. Because each of the processes is an instance of a sequential program with extensions to send and receive messages, the task of debugging concurrent applications presents all of the challenges of debugging traditional sequential applications. However, it also has the added challenge of tracking the simultaneous...but independent... execution of many different processes on different nodes and monitoring the messages exchanged between processes.

The Concurrent Debugger (DECON) is a source level debugger designed to shorten the debugging cycle of iPSC/2 application developers. It provides all of the sophisticated capabilities found in symbolic sequential debuggers. The Concurrent Debugger has capabilities that are valuable in detecting and fixing iPSC/2 application bugs both in the purely sequential logic of a single process and in the logic of communicating and synchronizing multiple concurrent and independent processes via message passing.

In the iPSC/2, each process is uniquely identified by the node id (nid) it is executing on and its assigned process id (pid). Thus, processes are specified by providing their (nid:pid) combination. A context is then defined by providing a list of one or more (nid:pid) pairs to the **context** command.

Another critical ability required to debug large concurrent applications is the ability to quickly and easily find any *"lost messages."* These are messages that are sent from one process to another, but, due to various programmer errors, are not received at their destination. Lost messages typically halt the progress of the computation and are very hard to find without special support from the debugging tool. The Concurrent Debugger allows you to inspect the system message buffers where all messages arriving at a destination node are stored until they are received by the proper destination process. Any lost messages would be stored here and can be detected by using the **msgq** (for message queue) command .

The **msgq** command not only lets you find lost messages, but also trace their cause. It does this by providing information on each message's originating process, intended destination, type, and length.

A complementary Concurrent Debugger command is **recvq**. This command lists requests to receive messages that have not yet been satisfied with the arrival of the corresponding message.

Together, the **msgq** and **recvq** commands allow you to quickly and easily detect most, if not all, of the message related bugs in iPSC applications.

The Concurrent Debugger also has capabilities that are valuable in detecting bugs in iPSC/2 program logic. These bugs can be in the purely sequential logic of a single process or in the logic of communicating and synchronizing multiple concurrent and independent processes. To help you detect the first kind of problem, the debugger lets you zoom in on any node and apply the full power of a high level source code debugger to a single process. To help with the second, more difficult problem, the Concurrent Debugger takes those capabilities found useful in the first case and extends them to the parallel environment of the iPSC/2.

Simulator

The Simulator is a program which creates a simulated iPSC/2 environment on the programmers workstation. It allows you to prototype his concurrent application in a controlled environment prior to execution on the iPSC/2. Use of the Simulator can significantly speed up the initial program development cycle and free the iPSC/2 for more advanced stages of development and production runs.

Hypercubes are extremely flexible supersets of parallel topologies such as loops, trees, meshes, and toroids. The hypercube offers an ideal environment for embedding parallel applications. Coding in standard FORTRAN and C languages, you can realize parallel constructs of your programs by "test-bedding" them on the iPSC/2 Simulator. The application interface is consistent between the iPSC/2 Simulator and the iPSC/2 System, allowing easy migration if the performance of an iPSC/2 System is desired. This also speeds the program development cycle by providing facilities for proving out code designs as well as locating programming errors.

Re-hosting the simulator on other multi-tasking operating systems is possible and the simulator is delivered in source code form to support such projects. A detailed engineering design specification is supplied for the systems programmer who wishes to re-host the iPSC Simulator or enhance and customize its feature set.

Some of the simulator features are:

- **FORTRAN or C** — The simulator executes hypercube programs written in either the C language or FORTRAN. It simulates the actions of the iPSC/2 node executive routines invoked by the user's programs.
- **Modification Not Required** — Programs do not have to be modified to run on the simulator.
- **Interactive Interface** — The simulator provides an interactive interface which simulates the iPSC/2 system's System Resource Manager commands. You begin simulation by issuing these commands to the simulator. In addition, System Resource Manager commands can be placed in a script file and directed to the simulator's standard input on the command line.
- **Interrupt Handling** — You can interrupt the simulated hypercube's process (or processes), check its status, and then restart the simulator from where it was interrupted...without ever exiting the simulator.
- **Messages** — Both error and trace messages aid in detecting errors in the use of host and operating system routines.

CHAPTER 3

iPSC/2 SOFTWARE ARCHITECTURE

INTRODUCTION

This chapter discusses the following topics:

- overview of the iPSC/2 software environment
- host software
- node software

These topics are divided into discussions of various interrelated subjects concerning iPSC/2 programming. Various programming hints are scattered throughout this chapter. Before you begin to program the iPSC/2, we suggest you read this whole chapter.

TERMINOLOGY

The iPSC/2 software consists of iPSC/2 commands invoked from the terminal and libraries with external C routines, external FORTRAN functions and external FORTRAN subroutines that may be called from C and FORTRAN programs. This manual uses the term **routines** to refer to the C routines, FORTRAN functions and FORTRAN subroutines.

OVERVIEW OF THE iPSC/2 SOFTWARE ENVIRONMENT

The typical iPSC/2 application operates in the software environment of the host and of the nodes and has a program that runs on the host and a program that runs on the nodes. The node application programs are created, compiled, and linked in the host UNIX environment. The host application programs are created, compiled, and linked in the host environment.

In a typical application:

- Commands issued from the host terminal allow access to the cube, give information about the cube, and control the cube I/O. These commands prepare the system to run your application program.

- The part of the application that is the **host program** executes in the UNIX environment as one or more processes, typically providing the initialization for the application and any necessary human interface. The host program can load the nodes with their individual programs and communicates with the node programs via messages.

- The part of the application that is the **node program** executes in each node's NX/2 operating system environment as one or more processes. These processes run concurrently. Typically these programs do calculations and exchange data via messages with other node processes, sending results back to the host process.

This section discusses important features of the host software that interfaces with the cube. It also covers cube partitioning and the node operating system. Topics in this section include:

- The Host
- Host Processes
  - File Servers
  - Commser
  - Lifeline
- Controlling Cube With Commands
  - Cube Sharing
  - Program Control
- Controlling Cube With Routines
- NX/2 Operating System

THE HOST

The iPSC/2 system is composed of the system resource manager (SRM) and the cube. The SRM serves as a gateway to other computers and workstations via an Ethernet connection and is connected to the cube via a Direct-Connect Module. The "host" computer may be a workstation on the network or an SRM itself.

The host serves several functions:

• Developing software	The development environment, including compilers, linkers, debuggers and UNIX software tools are on your host computer.
• Controlling hardware cubes that are connected to the SRMs on the network	Commands entered on the host terminal, or routines used in host programs, allow you to control and access the various cubes connected through SRMs to the network.
• Interfacing application programs	The host part of your our application program will run on your host computer. It generally handles initializing, human interface, and result reporting.
• Interfacing file input and output from node computers	The individual node computers that are part of the hardware cube(s) have no hard disk storage and no file systems. The node computers can write to and read from files on the host system.

Host Processes

Several host background processes are started automatically by iPSC/2 software. These processes are responsible for handling node-to-host message passing, node and host I/O, and cube network communications. This section discusses these system processes in more detail.

File Servers

File servers allow node programs to use standard I/O functions, such as read from and write to the host file system, and read from and write to the user's terminal.

The file servers are background host processes designed to handle I/O requests for the nodes and are started automatically by the *getcube* command (or routine). (Refer to the *Cube Sharing* section for information about *getcube*.) The number of servers that are started depends on the size of your cube as shown in Table 3-1:

Table 3-1
Number of Available Servers

Number of Nodes	Number of Servers
0 to 32	1
33 to 64	2
65 to 96	3
97 to 128	4

Each file server can open up to 100 files. This means that all the nodes serviced by one file server can open (and read from and write to) a total of 100 files at one time. For example, in a cube with 32 nodes, each node could open 3 files (3 x 32 = 96) at once. A cube with more nodes has more file servers, maintaining this average of possible open files/ node. Refer to the *Input/ Output* section for more information on how the file servers affect input and output.

Commser Process

The commser (communication server) host process is used to route host-to-node and host-to-host message requests. This background process is started by the administrator's cube- initialization command *bootcube*. There is a commser process on the SRM and also one on each remote host. The commser process on the remote host communicates with the commser process on the SRM for remote host to node communications. The Direct-Connect module device driver communicates with the SRM commser process. Some cube error messages mention commser. If these error messages persist, contact your system administrator.

Lifeline Process

The lifeline process provides information about user processes to the commserver process, and broadcasts the SRM status to all remote hosts on the Ethernet network. This background process is started by the administrator's cube-initialization command *bootcube* and exists only on the SRM. The lifeline process has two purposes. It informs the commserver process when any host process using the cube dies, permitting the commserver process to handle messages for that host process. It also detects when a file server dies. If a file server for a cube dies, then the cube is released, releasing associated nodes back to the available pool. Some cube error messages mention lifeline. If these error messages persist, contact your system administrator.

CONTROLLING THE CUBE WITH COMMANDS

An important feature of iPSC/2 software is its ability to do cube control (and obtaining cube information) via commands from the terminal.

Table 3-2 below gives a brief description of cube control and information commands.

Table 3-2
Cube Command Summary

Command Invocation	Description
attachcube [-c cubename]	Make the default or specified cubename the current attached cube.
cubeinfo [-a] [-s]	Return cube ownership information.
getcube [-c cubename] [-t cubetype] [-h srm]	Allocate a cube and make it the current attached cube.
killcube [-c cubename] [-p pid] [node [node...]]	Kill node processes.
load [-c cubename] [-p pid] [-H] [node..] file	Load a user process into the cube.
newserver [-c cubename]	Start a new file server for the specified cube.
rcc [[-cpp] [-h srm] [c_options]] filename *rf77 [[-cpp] [-h srm] [rf77_options]] filename* *rld [[-h srm] [ld_options]] filename* *ras [[-h srm] [as_options]] filename* *rar [[-h srm] [ar_options]] filename*	Remote iPSC/2 development utilities which are: the C compiler, FORTRAN compiler, linker, assembler, and librarian. These are available on the remote host only and allow the remote host to build node executable programs.
relcube [-c cubename] [-a]	Release a cube(s).
startcube [-c cubename] [-p pid] [node [node...]]	Start a process executing on the cube.
syslog [-o] [-e]	Send the output of a host process to the file server handling I/O from the nodes.
waitcube [-c cubename] [-p pid] [-f] [node [node...]]	Wait for process(es) on node(s) to complete.

Cube Sharing

Before a program can be run on the cube, nodes must be allocated for the exclusive use of a user. The *getcube* command allows you to allocate nodes. While there may be up to 128 physical nodes in the system, you may reserve some subset of these nodes for your use. This allows multiple users to "share" the cube by dividing the cube into "subcubes." These subcubes are completely independent and, once allocated, are not available to other users. The *relcube* command relinquishes the nodes allocated by *getcube* and makes the nodes available to other users.

A single user may use *getcube* to obtain several subcubes. The *attachcube* command and routine allow you to "attach" to a particular subcube and communicate with it. In this case, you must be attached to it. The *cubeinfo* command and routine allows you to see what cubes have been allocated and to whom.

Program Control

Node programs are developed and compiled on the host system. The *load* command on the host allows you to load a program on to the cube. *Startcube* will start the process running on the nodes, and *killcube* will kill those processes. The *waitcube* command will block waiting for completion or death of a process on the cube.

If the host system is not the SRM but rather a workstation on the network, then program development tools are provided for compiling and linking node programs. These commands are named the same as the tool on the SRM, with an "r" (for "remote") prefix. For example, cc on the SRM is *rcc* on the remote development environment. Using these tools on the remote development environment allows you to develop programs to run on the cube.

As explained earlier, file server processes are started when you issue a *getcube* command. These background processes field file access requests from the nodes. The file server processes inherit the shell environment of the process which started them, in this case, *getcube*. After you issue a *getcube*, the environment may change for some reason. To change the environment of the file server, you must issue a *newserver* command. This will kill the old file server(s) and start new one(s) in the new environment. More information is given in Chapter 4 of this manual.

CONTROLLING THE CUBE WITH ROUTINES

Cube control commands are available in a programmatic interface. This allows a program on the host or node to control the cube. For example, a host program can issue a *getcube* call and a *load* call. Table 3-3 below is a summary of cube routines.

Table 3-3
Cube Routine Summary

Routine	Host/Node	Synopsis	Description
attachcube	host	attachcube(cubename) char *cubename;	Attach to a cube and make it the current cube.
cubeinfo	host	cubeinfo(cubetable, numslots, global) struct cubetable *ct ; long numslots; long global;	Obtain information about allocated cubes.
getcube	host	getcube (cubename, cubetype, hostname, keep) char *cubename; char *cubetype; char *hostname; long keep;	Allocate a cube.
killcube	host/node	killcube(node, pid) long node, pid;	Clear out a process.
killproc	host/node	killproc(node, pid) long node, pid;	Terminate a process.
killsyslog	host	killsyslog()	Terminate *syslog* process.
load	host/node	load(filename, node, pid) char *filename; long node, pid;	Load a node process.
newserver	host	newserver(cubename) char *cubename;	Start a new file server for the specified cube.
relcube	host	relcube(cubename) char *cubename;	Release a cube.
setsyslog	host	setsyslog(stdfd) long stdfd;	Start the *syslog* program.
waitall	host/node	waitall(node, pid) long node, pid;	Wait for all the specified processes to complete.
waitone	host/node	waitone(node, pid, cnode, cpid, ccode) long node, pid; long *cnode, *cpid, *ccode;	Wait for a specified process to complete.

CONTROLLING THE CUBE WITH ROUTINES (continued)

Note that not all of the commands discussed are implemented in the programmatic interface. For example, the remote development tools are not available. However, there are some additional programmatic calls available. In addition to *killcube*, a program may terminate a process using *killproc*, or terminate a syslog process using *killsyslog*. Instead of *waitcube*, the programmatic interface provides *waitall* and *waitone*.

Some of these routines are available only on the host. For example, a *getcube* call may only be executed in a host program. Table 3-3 shows which calls are limited to the host.

NX/2 NODE OPERATING SYSTEM

The NX/2 (Node eXecutive/2) operating system exists in RAM memory on each node. This operating system is responsible for running your node programs. NX/2 has these features:

- Runs the executable programs produced by the host C or FORTRAN compilers as processes.
- Supports up to 20 time-shared processes per node using round-robin scheduling.
- Provides synchronous and asynchronous message sending and receiving services to and from other node and host processes.
- Supports UNIX-style input/output to the host file system and host terminal.
- Provides memory management automatically for processes.
- Provides each process with a hardware enforced protected memory space.
- Provides "exception" handling via interrupts. An exception is a condition that occurs during process execution that requires some attention but may not require that the process be killed. One example is attempting to access an invalid memory address.
- Provides debug support for the DECON Concurrent Debugger.

The NX/2 operating system is loaded on the nodes with a command available only to the system administrator. This command is called *bootcube*. Refer to Chapter 4 for the use of this command.

iPSC/2 PROGRAMMING OVERVIEW

This section provides information you must know before attempting to program the iPSC/2 system. It includes the following topics:

- Program Compilation On Host
- Host/Node Byte Order Differences
- Multiple Users
- Host File System
- Node Numbers and Process ID's
- Node Processes and Routines
- Message Passing
- Exception Handling
- Error Handling
- Input/Output

PROGRAM COMPILATION ON HOST

Node programs must be compiled and linked on the System Resource Manager. It is advantageous to build node programs on the SRM for two reasons:

- It offloads work from your workstation
- It minimizes porting development tools to other workstations

However, utilities are provided so that you can build programs on a remote workstation:

rcc	remote C compiler	***Note that these utilities have an "r" prefix to prevent conflict with local utilities.***
ras	remote assembler	
rf77	remote FORTRAN compiler	
rld	remote linker	
rar	remote library archiver	

Any files produced by the above utilities have an ".ipsc" appended.

For example, the following invocation:

cc -o node node.c -node	On system resource manager
rcc -o node node.c -node	On the remote workstation

produces:

node	On system resource manager
node.ipsc	On the remote workstation

HOST/NODE BYTE ORDER DIFFERENCES

A remote workstation may have a different internal byte order representation of integer and floating-point values than the node representation of these values. The iPSC/2 provides utilities for each remote workstation that convert *node* byte order to *workstation* byte order, and vice versa.

The node byte order is expected when sending a message from a remote workstation to the nodes, or from the nodes to the workstation. Therefore, it is necessary to use the byte swapping utilities when the host program on a remote workstation is sending a message. These utilities must also be used when receiving a message from the node before the message can be used.

An example of a host program using some of the byte swapping utilities is:

```
main( ) {
            •
            •
long detain[100];
long dataout[100];
            •
            •
      HTOCL(datain, 100)
      csend (NODE_TYPE, datain,
          sizeof(datain) , -1, NODE_PID);
            •
            •
      crecv (NODE_TYPE, dataout,
            sizeof(dataout) );

      CTOHL(dataout, 100)
            •
            •
}
```

These byte swapping utilities require the following two parameters:

- starting address of the message
- number of elements to swap

If the program's message is a structure of different types, you will to to use these utilities on each of the different elements in the structure.

MULTIPLE USERS

Cubes are allocated with the ***getcube*** command which includes a ***size*** switch allowing you to specify a subset of the available nodes.

Each subcube you allocate is independent of other subcubes and is protected from corruption by the node operating system. Subcubes are allocated on a "first fit" basis, where the allocation is the first contiguous set of nearest neighbor nodes that match your request.

The ***size*** switch is an optional parameter. Some notes on this parameter are:

- **Size Omitted** — If the cube size is omitted, the system allocates the largest available subcube. Thus, if the entire cube is available, then the entire cube is allocated.
- **Not A Power of 2** — If you request a number of nodes other than a power of two, the next greater power of two is allocated.
- **Not Available** — If the specified size is unavailable, no cube is allocated and an error message is displayed.

One application may own multiple subcubes. However, the application can only communicate with one subcube at a time. In order to communicate with a specific subcube, either a user or an application must be connected to it. When you are connected to a subcube, it is called ***"the current cube."***

Once you perform a successful ***getcube***, you are connected to that cube. The ***attachcube*** call or utility lets you connect to a different subcube. Applications that make multiple ***getcube*** requests must give a unique name to each subcube.

Although message passing between subcubes is not supported, a host application can serve as a gateway for communication between subcubes. For example:

- Ensure the application owns mutliple subcubes
- The application receives a message from one subcube
- The application then uses ***attachcube*** to obtain a ***different*** subcube
- The application then sends the message on to the new subcube

The actual physical nodes allocated to a subcube are transparent to the user. In an application, node numbering is always 0-(N-1), where N is the number of nodes in the partition minus one.

HOST FILE SYSTEM

The iPSC/2 has an extended file system access to node programs. Standard C and FORTRAN I/O calls can be made from node programs. This includes reading and writing host files as well as standard input and output. An example of allowable C and FORTRAN code on the nodes is:

```
main(  )
{
          .
          .
     fd=open("foo", RW);
     read(fd, data, 100);
          .
          .
}
```

```
     PROG TEST
          .
          .
     OPEN(9, STATUS='OLD', FILE=FOO)
     READ(9, *)  (DATA(I),  I=1, 100)
          .
          .
     END
```

When you allocate a cube with ***getcube***, a file server which handles I/O requests is automatically started. Node programs may access files with either absolute or relative pathnames. If using relative pathnames, the current working directory of the file server is used for seraching. The current working director of the file server is established with ***getcube***. If you change working directories after invoking getcube, the new current directory does *not* get passed on to the file server *unless* you invoke the ***newserver*** command. The ***newserver*** command actually kills the original file server and starts a new one, using the current working directory.

If you want to redirect the standard input, standard output, or standard error of the file server, then ***newserver*** and ***getcube*** must be redirected. For example, invoke:

getcube>outfile

This causes the standard output of the file server and the nodes it serves to be redirected to ***outfile***.

As with any UNIX process, when you log-out, a hangup signal is sent to the file server which causes the file server to terminate. A subcube cannot remain allocated without a file server. When the system detects a file server termination, it releases the subcube. Maintaining ownership of a subcube through a logout can be done by using ***nohup***. For example, invoking:

nohup getcube

causes the file server to ignore the hangup signal.

NODE NUMBERS AND PROCESS ID's

Message-sending routines specify the destination of the message. This destination is defined by a node number and a process id. Each node, and the host, is assigned a node number. Node numbering starts at 0 and runs sequentially to one less than the number of nodes requested. The node number of the host is equal to the number of nodes in the attached cube. If you request 8 nodes, the node numbers for the host would run 0-7. The node number of the host would be 8.

Each process running on a node, and message-passing processes running on the host is given a process id (pid). A process id may be 0 or any positive integer and is assigned differently for processes on the host and the nodes. One node may not have more than one process with the same pid but different nodes may have processes with the same pid.

The *load* command or routine assigns process ids for node processes automatically. (A process on a node may determine what its own pid assignment was via the *mypid* routine.)

The *setpid* routine, called only from a host process, allows the user to set the host process id via an input parameter that is the user's pid choice.. *Setpid* sets the host's cube process id, not to be confused with the host's UNIX process id that is automatically assigned by UNIX. If your application requires multiple host cube processes, each process must be assigned a unique cube pid via *setpid*. If you attempt to assign the same pid to two host processes, an error will occur: "setpid: Pid already in use"

NOTE Host programs that use a cube must be attached to a cube (via *getcube* or *attachcube*). Then use the *setpid* routine before the host program attempts to use any routines to communicate with the cube.

MESSAGE PASSING

The iPSC/2 is a distributed memory machine. Nodes communicate with each other and with the host via message passing. A message may be up to 256 Kbytes long if data is being passed to or from the host. It may be of unlimited length if it is being passed from node to node. Messages may be any format or data type. A sending process may send a message of a particular data type. It is your responsibility to ensure the receiving process expects a message of that data type.

A message can be sent from one process and received by one or more processes on the same node, or on different nodes. Messages may also be sent from one or more processes on the host to one or more processes on the node, and vice versa. From a programming point of view, sending a message sends a copy of the contents of a buffer from one process to another. Note that the buffer variable may be of any data type including the C data type "struct", allowing any amount of data to be sent in one variable. Note also that messages of 0 length can also be sent. These may be used as signals to coordinate node programs and have the advantage of very short transmission time.

The receiving process(es) may be on other nodes, on the host, or even to the same node. The buffer variable is set to the value(s) in the sending process. The message-receiving routine places the copy of the sent value(s) into a local buffer. Messages can be sent and received by both host and node processes. Note that hosts have the same node number, equal to the number of nodes in the cube.

A message may be sent to other nodes in the cube or to a specified node. It may be sent to a particular process pid or all process pids. For example, you can attempt to send a message to process 27 on node 4, or to any process on node 0.

When used in your application, messages may need to be synchronized so that your concurrent programs will have the data they need at the proper time. Refer to the *Messages in the Concurrent Environment* section to learn the features that permit message synchronization.

To present a detailed description of message-passing, the remainder of this section covers the following topics:

- Message Characteristics
- Messages in the Concurrent Environment
- Receiving Messages
- Interrupts from Messages
- Message Control
- Message Information
- Syntax of Message Routines

Message Characteristics

Message have a type, length, and an ID. These are parameters used in message-passing routines.

- **type**

  The type parameter is a tool you can use to identify a message. The data type of this parameter is long and the valid range of values for it is 0 or any positive integer. You may want to identify a message as a certain data type, or as a flag that the message contains some particular information. For example, messages of type 0 could indicate the message contains a character array while messages of type 1 could indicate the message contains a floating-point array. Another example is that messages of type 0 include the ages of children while messages of type 1 include the ages of parents.

  Several of the message-passing routines permit the use of this message type selector parameter. The type parameter can be used to indicate messages of one type, of all types, or of a group of types. This feature allows processes to receive messages of any type when you want to receive messages in first-come first-served order regardless of type, or to receive messages of a set of types when the process does not care in which order messages of these types are received. Refer to *Appendix B "TYPESEL MASK"* in the *iPSC/2 C Programmer's Reference Manual* or *Appendix A "TYPESEL MASK"* in the *iPSC/2 FORTRAN Programmer's Reference Manual* for more information.

  Send and receive message routines use buffer variables of some data type to send and receive messages of some message type. It is convenient to use a received buffer variable of the same data type as the sending buffer variable data type, but this is not necessary.

  For example, you might wish to send a text message between processes on two nodes. You place the text into a variable of data type *array of char* (or *array of character* for FORTRAN) and assign this message the message type 7 in the message-sending routine call. The message-receiving call must be set to receive messages of type 7 into a variable of the same data type (*array of char or character*).

  If the receive and send data types are different, you must take into account the different memory storage syntax of these data types to be sure you interpret the message correctly. For example, if you sent a message placed in a buffer of data type *array of long* and received it into a buffer of data type *array of short*, values placed into the sending array at one array index (e.g. sendbuf[7])would be retrieved from the receiving array at a different index (perhaps recbuf[14]).

Message Characteristics (continued)

- **length** — All messages have a message length in bytes. The message-sending routines will send no more than the specified message length number of bytes. (Zero length messages may also be sent.) Message-receiving routines will place no more than the specified message length number of bytes into the receive buffer. The iPSC/2 *infocount* routine returns the number of bytes in a received message. Note that the value returned by *infocount* may be different than the length parameter value. For example, if a receiving buffer length is 10 and a message of size 15 comes in, then 10 bytes get copied to the buffer, 5 get thrown out, and *infocount* returns 15.

- **message id** — Asynchronous (see the following section) message-passing routines return a unique message id (a positive integer) identifying each message. This id allows you to get information about the message.

Messages in the Concurrent Environment

The iPSC/2 message routines contain a number of features that allow messages to work efficiently in the concurrent environment. The remainder of this section explains some of those features. Through proper use of these various message features you can optimize your application to allow as much time as possible in calculation and as little as possible in sending, receiving or waiting for messages.

In the iPSC/2 system, messages are passed synchronously or asynchronously:

- **Synchronous** — A call to the synchronous message-passing routines (*csend* and *crecv*) blocks until the message is sent or received before returning and allowing program execution to continue.

- **Asynchronous** — A call to the asynchronous message-passing routines (*isend* and *irecv*) returns immediately and does not block until the message is sent or received. The program continues execution whether or not the message is sent or received. The *isend* and *irecv* routines return a message id identifying the particular message that has been sent or the receive request. This message id can be used as a paramter in the *msgwait* and *msgdone* routines to check for completion of the operation.

Receiving Messages

Once a message is sent, it becomes a pending message that may or may not be received. The *cprobe* (and *iprobe)* routine allows a process to determine if a message of a particular message type (or range of message types) is pending for it. (Note that a message sent to a particular node number and pid can only be received by that pid on that node.) Once these routines indicate a message is pending you may use the *crecv* or *irecv* routine to get the message.

- **Determining if a specific message type is pending**

  There are a number of "info" calls to allow the process to get more information. *Cprobe (*or *iprobe)* takes a type(s) parameter and checks for pending messages of that type(s). *Cprobe* is a synchronous call, blocking the process until a message of that type(s) is pending. *Iprobe* is asynchronous, reaturning a 0 or 1 value if found.

- **Determining if a message has been sent or received**

  The *msgwait* and *msgdone* routines can be used to determine if an asynchronous message, identified with a message id, has been sent or received. The return of 1 from *msgdone* or a return from *msgwait* (after an *irecv* or an *isend* call) indicates that the message has been sent or received. When *msgdone* returns a 1 the send or receive is complete and the message id is no longer valid in *msgwait* or *msgdone* calls.

Note that if you use a *msgwait* call with the same message id as a previous *msgdone* call that has returned 1 (success), an error ("Invalid message id error") may occur. If the *msgdone* call has returned a 0 however, *msgwait* can be used with no error.

NOTE Do not use an *cprobe* call after a *irecv* call to probe for a particular message. The *irecv* call will get the message, removing it from a message queue and the *cprobe* call will wait indefinitely trying to find that message. Use *cprobe* before *irecv*.

Interrupts from Messages

The *hrecv* routine can be used to permit node (not host) processes to be interrupted upon receipt of a message.

Message Control

The *msgcancel* and *flushmsg* routines can be used to cancel pending asynchronous messages. *Msgcancel* is useful in debugging applications. Your program can contain a timing loop using *msgdone* following an *irecv* call. If *msgdone* does not return 1 within the specified time, a *msgcancel* call can be used to cancel the receiving of the message. The *flushmsg* routine can be used to get rid of extraneous messages.

Message Information

The *infocount*, *infonode*, *infopid*, and *infotype* routines can be used to get information about pending or received messages.

Syntax of Message Routines

Table 3-4 gives the syntax for the various C message passing routines. The FORTRAN versions provide identical functionality and are very similar in syntax. Refer to the ***iPSC/2 C Programmer's Reference Manual*** and the ***iPSC/2 FORTRAN Programmer's Reference Manual*** for details of the routines and procedures.

Table 3-4
C Message Routine Summary

Routine	Host/Node	Synopsis	Description
cprobe	host/node	cprobe(typesel) long typesel;	Wait for a message to arrive.
crecv	host/node	crecv(typesel, buf, len) long typesel; char *buf; long len;	Receive a message and wait for completion.
csend	host/node	csend(type, buf, len, node, pid) long type; char *buf; long len, node, pid;	Send a message and wait for completion.
flushmsg	host/node	flushmsg(typesel, node, pid) long typesel, node, pid;	Flush specified messages from the system.
hrecv	host	hrecv(typesel, buf,len, proc) long typesel; char *buf; long len; void (*proc)( );	Provide user-written exception handler for receive traps.
infocount infonode infopid infotype	host/node	long infocount() long infonode() long infopid() long infotype()	Returns information about a pending or received message.
iprobe	host/node	long iprobe(typesel) long typesel;	Determine whether a message of a selected type is pending.
irecv	host/node	long irecv(typesel, buf, len) long typesel; char *buf; long len;	Receive a message.
isend	host/node	long isend(type,buf, len, node, pid) long type; char *buf; long len, node, pid;	Send a message.
msgcancel	host/node	long msgcancel(id) long id;	Cancel a send or receive operation.
msgdone	host/node	long msgdone(id) long id;	Determine whether a send or receive operation has completed.
msgwait	host/node	msgwait(id) long id;	Wait for completion of a send or receive operation.

ERROR HANDLING

Host and node programs may have errors at execution. Some of these errors will terminate the node's process while other errors will not terminate the process.

- **Node Program Execution Errors**

  As with any program, node programs may have errors at execution. If an error is encountered in the execution of the standard iPSC/2 routines, then the node's process is terminated and an error message is sent to standard error on the host via the node's file server process.

- **Non-Process-Terminating iPSC/2 C Routines**

  There are additional forms of the iPSC/2 C routines. An error in the execution of these routines will not cause the host or node process to terminate. These routines have the same name as the standard form with the addition of an underscore ("_") in front of the routine name. For example, the non-process-terminating form of the C routine "getcube" is "_getcube".

  These routines also return the value of -1 if an error is encountered or a zero or positive value if the execution is successful. The system variable *errno* is set to contain the error code in case of error. Include the <errno.h> and <cube.h> files to use these error numbers.

EXCEPTION HANDLING

Various other errors and conditions needing attention are possible at node program execution time. For example, trying to access an invalid memory address will cause an exception. The *handler* routine allows you to trap for a particular exception type and begin execution of your own exception handling routines. Refer to the description of *handler* in the *iPSC/2 C Programmer's Reference Manual* and the *iPSC/2 FORTRAN Programmer's Reference Manual.*

INPUT/OUTPUT

This section discusses the input/output facilities for the host and node programs and covers:

- Host I/O
- Host I/O Redirection
- Node I/O
- Node I/O Redirection

Host I/O

The host handles input and output from both host and node programs. iPSC/2 host commands or routines called from host processes may be used to control host and node I/O. Several routine calls duplicate host commands allowing I/O features to be changed from within a host program as well as from host terminals.

Host process I/O is sent by default to the terminal or host file system as determined by the host program. Standard C and FORTRAN I/O calls may be used in your host program. You may also use the standard UNIX redirection of host I/O.

Host I/O Redirection of Node I/O

Node I/O is handled by a file server process (started by *getcube*) running on the host. The standard output and standard error of this file server is the terminal, but these may be redirected to a file with UNIX redirection when *getcube* is invoked, or with the *newserver* command or routine. For example, after the command **getcube > myfile**, subsequent output to the node's standard output will go to the file *myfile*.

The *syslog* command (used as a filter with a pipe) or the *setsyslog* routine is used to redirect the standard output from a host process to the same location as the standard output of the node I/O. Thus, if node I/O is writing to a file, host I/O can be sent to the same file. For example, you may wish to send the standard output from your host program *myhost* to the same file that is receiving your node programs' standard I/O. These two commands will accomplish this:

```
getcube > myfile
myhost| syslog
```

Node I/O

You may use the standard I/O FORTRAN or C routines in the node programs. The I/O requests are sent to background file server processes on the host.

NOTE Using calls in node programs that request standard input requires special handling. This is due to the fact that the host shell process and the node process are in a race condition as to which one gets the characters from the terminal.

To have a node request standard input, put the shell process to sleep. This can be done by the commands:

hostprog; **sleepprog**

Where *hostprog* is your host application program that loads the node program that requests standard input and **sleepprog** is the following program:

```
main()
{
    while(1) sleep(1000);
}
```

This *sleepprog* causes the shell to sleep until an interrupt character (usually "Del") is entered. You must enter this interrupt to continue using UNIX (restoring your shell) after your *hostprog* is finished.

Similarly, only have one node program at a time request standard input. If more than one node process requests standard input at the same time, the input characters will be distributed amongst the nodes in an unpredictable fashion.

The *getcube* invocation starts the file server(s). As mentioned in the File Server section, the file server process inherits the shell environment at the time when *getcube* is invoked and does not change this environment. Part of this environment is the current directory. This has some important programming implications as explained in the next paragraphs.

As mentioned, the standard output and standard error of the file server is the terminal, and these may be redirected to a file with UNIX redirection when *getcube* is invoked. For example, after the command **getcube > myfile**, subsequent output to the node's standard output will go to the file *myfile*. This redirection has the file server send node output to the file *myfile* in the current directory. Implicitly, the current directory is determined by the current directory where *getcube* (that started the file server) was invoked.

Now examine what happens when you run your node programs that produce output to the file *myfile*. Whatever the actual current directory when the node programs are invoked (e.g. enter: **load nodeprog**), the output of the node programs will go to the old current directory in effect for the file server. If you have changed directories the output will still go to that original file *myfile*.

This previous current directory situation also applies to files named in I/O routines in your node programs.
That is, if a path to a file implicitly refers to the current directory, it will be the old current directory in effect when *getcube* was invoked. For example, in this node program C fragment:

```
fptr = fopen("somefile","w");
fprintf( fptr,"Hello There!");
```

and in FORTRAN:

```
      open (9, file = 'somefile')
      write (9,10) 'Hello There!'
10    format (a12)
```

The text "Hello There!" will be sent to the file *somefile* in the current directory when *getcube* was invoked.

Node I/O Redirection to New Directory

The *newserver* command allows you to redirect I/O to a new current directory.

The *newserver* command (or routine) creates a new file server process for the node I/O and kills the existing file server process for the node I/O. The new file server process will have the current shell's environment and thus any I/O that was previously directed relative to an old current directory will be directed relative to the present current directory. For example, with these commands:

```
cd $HOME
getcube > myfile2
load nodeprog
cd newdir
newserver > myfile2
load nodeprog
```

The standard output from *nodeprog* will go to *myfile2* in the *$HOME* directory after the first run of *nodeprog* (**load nodeprog**). The second time *nodeprog* is run the output will go to *myfile2* in the *newdir* directory because of the *newserver* invocation. Without *newserver*, the standard output from the nodes would still go to myfile in the *$HOME* directory. Note that *newserver* can also use UNIX redirection (e.g. **newserver >anyfile**).

CHAPTER 4

USING THE iPSC/2 SYSTEM

INTRODUCTION

This chapter discusses the following topics:

- The directory structure and the files that make up the iPSC/2 software.
- Examples of the iPSC/2 commands you use to get information about the cube and control the cube.
- Compilation and loading of C, FORTRAN and iPSC/1 source programs for use on the iPSC/2.
- Compiling, linking, and running an example C program.

iPSC/2 SOFTWARE DIRECTORIES

The iPSC software resides in several directories as explained below.

/usr/bin	This directory contains the executable cube manager software, including cube manager commands such as *load* and *getcube*. The default path search for the system resource manager includes this directory.
/usr/ipsc/conf	This directory and its sub-directories contains the hardware configuration "cubeconf" file. This file contains information about your system configuration and is read by various iPSC/2 programs. Refer to the iPSC/2 System Administrator's Guide for details.
/usr/ipsc/examples	This directory contains C and FORTRAN programming examples.
/usr/ipsc/lib	This directory contains NX/2 (in a file called *nx.b*), the C and FORTRAN system resource manager interface libraries and the C and FORTRAN node interface libraries used to keep compatibility with the iPSC/1 programs. Also in this directory are four ".def" files. The *chost.def* and *cnode.def* files contain external declarations for the C library routines used to keep compatibility with iPSC/1 cube programs. The *fhost.def* and *fnode.def* files contain the FORTRAN external definitions.
/usr/ipsc/sim	This directory contains iPSC/2 Simulator source and object code.
/usr/ipsc/log	This directory contains system log files.
/usr/lib	This directory contains the iPSC/2 routine libraries *libhost.a* and *libnode.a*.
/usr/ghlang	This directory and its sub-directories contain the various files for the Green Hills FORTRAN Compiler.
/usr/include	This directory has the C include file *cube.h* and the FORTRAN include file *fcube.h*. Be sure to include the C file in your C programs and the FORTRAN file in your FORTRAN programs.
/usr/ipsc/IPSC2	This file lists all the iPSC/2 files.
/usr/ipsc/install.ipsc	This file is used in software installation.
/usr/ipsc/diag	This directory contains the cube diagnostic software.
/usr/ipsc/src/rhost	This directory has the remote host software to be installed on your remote workstation.
/lib	This directory contains libraries and object files for the iPSC/2 compilers.
/bin	This directory contains the iPSC/2 compilers.

Figure 4-1 diagrams the iPSC/2 software directories.

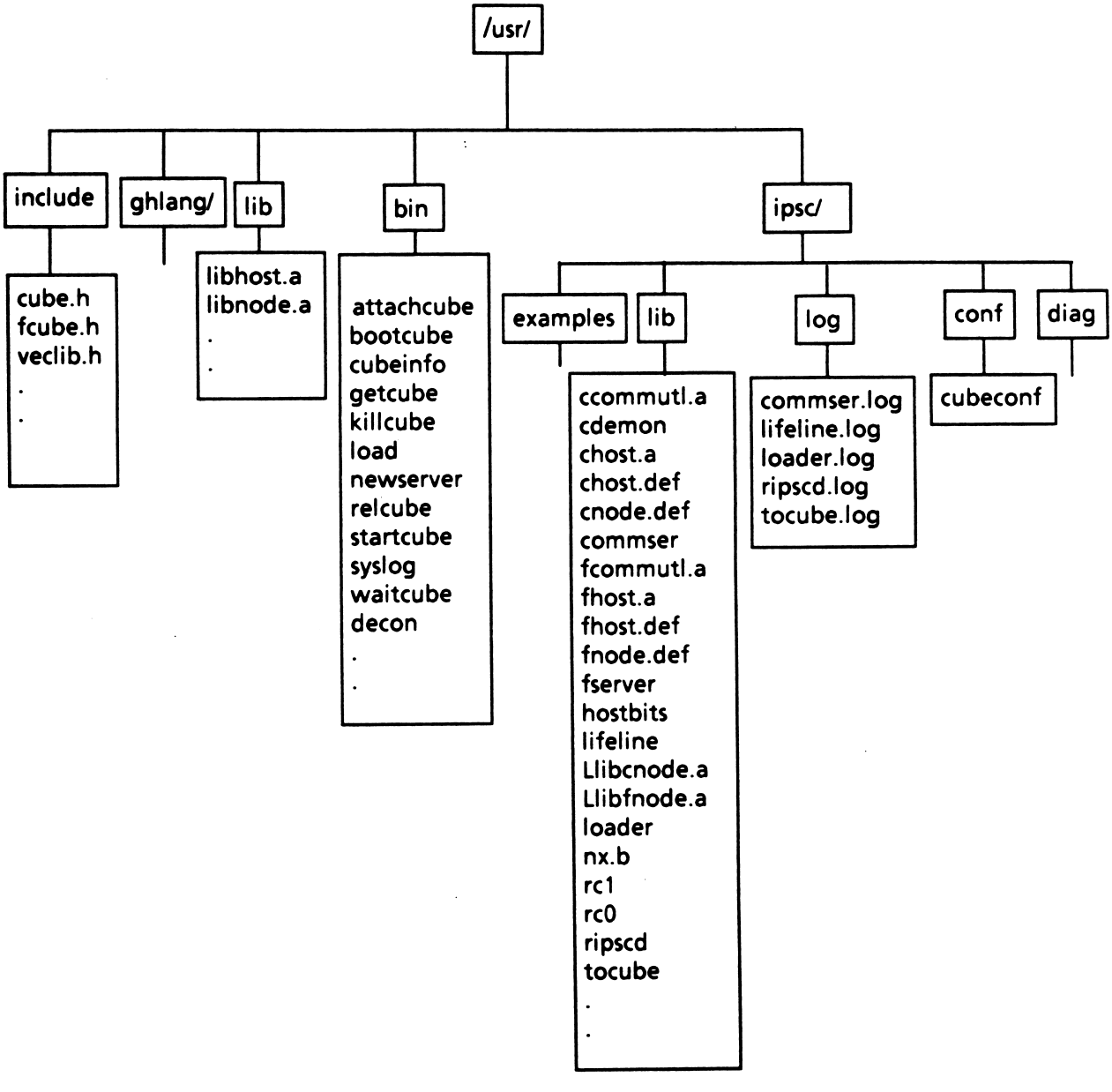

Figure 4-1
iPSC/2 Software Directories

CUBE SYSTEM ADMINISTRATION

This section discusses the administration of the cube.

The system administration command *bootcube* must be run before the cube can be used. This command resets the entire cube and reloads the NX/2 operating system into each node. Only the system administrator can run this command. Ordinarily, this command only needs to be run once when the system is first brought up. This command has a number of options and is described in the iPSC/2 System Administrator's Guide. Usually the cube is initialized by:

Enter: **bootcube**

CUBE CONTROL COMMANDS

This section shows you some examples using the iPSC/2 cube commands.

NOTE The TTY environment variable is used by several cube commands and must be set before you use the cube. If you use a Bourne shell, your default *.profile* file sets this variable for you, but if you do not use this default *.profile* file:

Enter: **TTY = `tty`**
export TTY

where 'tty' will get the name of your terminal, such as: /dev/ttyT2.

OR

If you use a C shell your default *.login* file also sets the TTY variable for you, but if you do not use this default *.login* file, set the environment variable TTY with the following commands:

Enter: **setenv TTY `tty`**

When using UNIX *sysadm* command to make users the default, *.profile* and *.login* files will contain the proper TTY setup.

Note that the character on each side of the "tty" in the above commands is a reverse accent mark, not an apostrophe or single quotation mark.

Getcube Command

Enter: **getcube -c fred -t8**
getcube -c wilma -t8 > cube.out

The *getcube* command reserves one or more logical cubes for your use. You may name your cube with the *-c* option. In this case you have named your first cube "fred" and the second cube "wilma". You may reserve and name more than one cube, each with a different name and each with its own set of nodes, but you can use only one cube at a time. (The *attachcube* command allows you to switch between your different cubes.) The *-t* option selects how many nodes you are reserving. If you enter *-t 8*, 8 nodes are reserved. The second *getcube* invocation reserves another 8 nodes. The *getcube* routine has the same functionality and may be called from your host program. Your currently attached cube is the last cube defined. In this case it is "wilma" and thus you have 8 nodes available to use.

If the output from the *getcube* command is redirected, as in the second example, the standard output from the node programs is directed to that file instead of your terminal screen. In the example shown, all cube standard output will go to the file *cube.out*.

NOTE You may request the reservation of any number of nodes, but *getcube* always reserves a number of nodes equal to an integral power of two with each call, if these are available. For example, **getcube -t1** reserves 2^0 (1) and **getcube -t16** reserves 2^4 (16). Non-integral choices reserve a number of nodes equal to the next highest power of two that is larger than the number requested. For example, **getcube -t9** will reserves $2^4 = 16$ nodes since $2^3 < 9$ but $2^4 > 9$. If you had a cube with a total of 64 nodes, **getcube -t33** would reserve the whole cube!

Note also that although the nodes are reserved you may not use them. If you try to load a process into a node greater than the number you requested, an error will occur. Refer to the *getcube* command in the iPSC/2 C (or FORTRAN) Programmer's Reference Manual for more information.

Getcube will report if your allocation was successful or not. If you asked for too many nodes, you will get the message: "cube type not found". In this case, request fewer nodes.

Attachcube Command

Enter: **attachcube -c fred**

The *attachcube* command changes your currently attached cube to the named cube. Now the currently attached cube is "fred" with 8 nodes.

Cubeinfo Command

Enter: **cubeinfo -s**

(typical display)

```
% cubeinfo -s
CUBENAME     USER      SRM       HOST      TYPE     TTYS
defaultname   ray      klingon   klingon   d3m4     ttyT3
demo          dre      klingon   muon      d4       REMOTE
life          michael  klingon   ruby      8        REMOTE
```

The *cubeinfo* -s option shows the cube usage information for the hardware cubes connected to the SRMs on the network. This display shows that users "ray", "dre", and "michael" have each reserved a logical cube (on the SRM klingon) from different hosts.

CUBENAME- the name given to the cube when it was allocated with *getcube*

USER - the user id of the owner of the cube

SRM - the name of the system resource manager

HOST - the host system from which *getcube* was invoked

TYPE - the size and type of cube displayed in a special syntax. The "d3m4" displayed indicates $2^3 = 8$ nodes of the "memory", 4 Mbyte per node type. An integer only indicates the number of nodes reserved.

TTYS - the terminal from which *getcube* was invoked. A cube could have multiple TTYs if multiple terminals are attached to the same cube (via *attachcube*). Remote means that the TTYs are not local to the host where the command was invoked.

The sum of all the nodes reserved on the SRM's hardware cube is naturally equal to or less than the total number of nodes it has available. In the above display, 32 nodes are already partitioned on the SRM named "klingon". If the hardware cube connected to klingon has 64 nodes, you can still reserve 32 nodes for your use in one or more logical cubes.

For further information, refer to *cubeinfo* in the *iPSC/2 C Programmer's Reference Manual*.

Load Command.

Enter: **load -c barney -p 10 nodeprog**

The *load* command loads an executable file onto the one or more nodes of your cube and assigns a process id number to the program to be run on the nodes. In this example, *load* loads all the nodes (by default) of cube "barney" (**-c barney**) with the executable file *nodeprog*, giving each process running on each of the nodes pid number 10 (**-p 10**). By default the *load* command instructs the nodes to start execution. The *load* routine permits you to perform the same actions from a host or node program.

Refer to the iPSC/2 C Programmer's Reference Manual for additional options of the *load* command.

Killcube Command

Enter: **killcube -c barney**

This command kills all processes on the cube **barney** and flushes all messages related to those processes. Use this command (or routine in your host program) to prepare the cube for your next program. For more information about *killcube* refer to the iPSC/2 C Programmer's Reference Manual.

Relcube Command

Enter: **relcube**

This command releases the cube you are currently using, allowing other users to use the nodes. Whenever you are finished with your cube(s), use this command (or routine in your host program) to release it. For more information about *relcube* refer to the iPSC/2 C Programmer's Reference Manual.

Newserver Command

Enter:

```
cd $HOME
getcube -c dragon -t16 > myfile1
load nodeprog
cd newdir
newserver > myfile2
load nodeprog
```

The *newserver* command (and *newserver* routine) starts a new file server process for handling node I/O and kills the old file server process that handled the node I/O.

In this example a cube named "dragon" is reserved and its standard output is redirected to *myfile1*. The standard output from the hypothetical *nodeprog* will go to *myfile1* in the *$HOME* directory after the first run of *nodeprog* (**load nodeprog**). The second time *nodeprog* is run, the node output will go to *myfile2* in the *newdir* directory because of the *newserver* invocation. Without *newserver*, the standard output from the nodes would still go to *myfile1* in the *$HOME* directory.

Syslog Command

Enter:

```
getcube -c dragon -t16 > myfile
 myhost | syslog
```

The *syslog* command (and *setsyslog* routine) redirects host process standard output or standard error to the same destination as the node process file server I/O redirection. Note that *syslog* is used as a filter and is the end of a UNIX pipe. In this example, the output from cube "dragon" is redirected to *myfile*. The standard output from the host program *myhost* is redirected to the same file *myfile*, rather than the host terminal, because of the *syslog* invocation. The host output will be interleaved with node output in the file.

Refer to the iPSC/2 C Programmer's Reference Manual or the iPSC/2 FORTRAN Programmer's Reference Manual for more information about the various commands discussed.

iPSC/2 COMPILATION AND LINKING

This section explains how to compile and link your C and FORTRAN host and node programs and how to use the UNIX *make* command to help you do this.

C Source Programs

Use the cc command to compile and link your host and node C programs. Be sure to include the include file *cube.h* in your C host and node source programs (**#include <cube.h>**). The compilation and linking are done as with any C programs with the addition of the libraries *libhost.a* and *libnode.a* and the standard *libsocket.a* and *libld.a* libraries. These are accessed from the *cc* command with the **-host** or the **-node** option.

To compile and link your host program:

cc -o *hostexec host1.c host2.c ...* **-host**

where:

hostexec	is the resulting executable host program.
host1.c host2.c ...	are one or more host source code files to be compiled and linked

To compile and link your node program:

cc -o *nodeexec node1.c node2.c ...* **-node**

where:

nodeexec	is the resulting executable node program.
node1.c node2.c ...	are one or more node source code files to be compiled and linked

The executable node program(s) will not run on the host. It is loaded and run on the cube by the *load* command or routine. Remember that each node program is itself a main program and that you may have many different node programs, possibly more than one on each node.

FORTRAN Source Programs

The iPSC/2 version of the f77 FORTRAN compiler has been enhanced to provide a common tool for vectorizing, compiling, and linking FORTRAN programs for the iPSC/2. iPSC/2's f77 also includes a powerful extention to *ld* for the iPSC/2-VX, *vxld*. The FORTRAN vectorizer, VAST-2 is also part of f77. As with all versions of f77, f77 can be used to link object modules into executable programs by invoking *ld*, the UNIX link editor. The functions and switches supported by the iPSC/2 version of f77 are described on the following pages.

Use the *f77* command to compile and link your host and node FORTRAN programs. This command invokes the Green Hills FORTRAN compiler and the system linker, ld. Be sure to include the include file *fcube.h* in your FORTRAN host and node source programs (**INCLUDE '/usr/include/fcube.h'**). The compilation and linking are done as with any FORTRAN programs with the addition of the libraries *libhost.a* and *libnode.a* and the standard *libsocket.a* and *libld.a* libraries. These are accessed from the *f77* command with the **-host** or the **-node** option.

To compile and link your host program:

f77 -o *hostexec host1.f host2.f ...* **-host**

where:

hostexec	is the resulting executable host program.
host1.f host2.f ...	are one or more host source code files to be compiled and linked

To compile and link your node program:

f77 -o *nodeexec node1.f node2.f ...* **-node**

where:

nodeexec	is the resulting executable node program.
node1.f node2.f ...	are one or more node source code files to be compiled and linked

The executable node program(s) will not run on the host. It is run on the cube by the *load* command or routine. Remember that each node program is itself a main program and that you may have many different node programs, possibly more than one on each node.

Green Hills Compiler Invocation Options

Several Green Hills compiler invocation options are frequently useful when compiling your FORTRAN and C iPSC/2 programs. These options are **-c**, **-v**, and **-o**. The **-c** and **-o** options behave in the same way as the *cc* compiler options of the same name. The **-v** option causes the compiler to print out the program name and command line arguments as it runs each subprocess. You may use this to see the location of the libraries. Refer to the following pages and the *Green Hills' C Language Reference Manual* and the *Green Hills' FORTRAN Language Reference Manual* for more information on the compiler invocation options.

Compiling

The Green Hills compilers support a variety of switches to handle special cases. These switches are described in the back of the Green Hills FORTRAN Manual. In general, the switches for f77 also apply to the Green Hills C compiler, *cc*. One switch has been added for the convenience of compiling programs for the Weitek 1167 (the SX option).

-sx compile for the 1167. (Note: Because the 1167 and 387 use different registers to return results, an executable program should contain compiled modules for only one processor.)

Linking

f77 uses the standard UNIX convention of searching for libraries in the directory */usr/lib*. The switches below link in the described libraries:

-lhost	the host communication and utility routines
-lld	the loader for loading the nodes
-lsocket	the socket library needed for fileio
-lnode	the node communication and utility routines
-lsxnode	the 1167 version of communication and utility routines
-lvx	the microcode for the veclib routines
-lvxvec	the 387 library of veclib routines executing on VX
-lvxdbvec	the 387 debug library of veclib routines executing on VX
-lvxsxvec	the 1167 library of veclib routines executing on VX
-lvxsxdbvec	the 1167 debug library of veclib routines executing on
-lsvecx	the compatibility veclib routines executing on 1167
-ldbvec	the compatibility debug veclib routines executing on 387
-lvxsim	the compatiblity microcode routines
-lvec	the compatibility veclib routine executing on the 387

The following flags abbreviate linking in specific libraries:

-host	is equivalent to -lhost -lsocket -lld
-node	is equivalent to -lnode
-sx	links in all 1167 libraries
-vx	is equivalent to -lvx
-vec	is equivalent to -lvec
-vecdb	is equivalent to -lvecdb

Vectorization

Files with the extension *.v* will be considered as files to be preprocessed by VAST-2 to produce *.f* files which will then be compiled by f77. All VAST-2 flags will be recognized by f77. (On those systems without VAST-2, f77 will abort when trying to execute VAST-2.) Further details about VAST-2 and its switches and flags are given in the *VAST-2 User's Guide*.

-double	use the double precision routines for vectorization (the default)
-single	use the single precision routines for vectorization
-both	use double and single precision routines for vectorization
-space = xxx	specifies xxx 4 byte words will be used by vast2 for vector temporaries
-lston = abc	specifies the abc listing options are to be on
-lstoff = def	specifies the def listing options are to be off
-vpfast	the 4096 4 byte words of static memory will be used by vast2 for vector temporaries

Linking for the VX - vxld

On the iPSC/2-VX there are two different memories: node memory on the node board, and VX memory on the vector board. Vector operands for the VX must reside in VX memory. To simplify allocating data to VX memory, the utility *vxld* will produce a link directive file to *ld* specifying where data will reside. The *vxld* switches that take *arg* as a parameter take a list of object names (.o files) as input. Each object file consists of possibly four sections:

.text	executable code
.bss	unintialize data
.comm	data initialized as zero
.data	initialized data

Each of the switches is used to place sections either in the VX memory or the node memory.(.text sections always reside in node memory.)

-b arg	puts .bss sections in VX memory
-c arg	puts .comm sections in VX memory
-d arg	puts .data sections in VX memory
-e arg	puts .bss, .comm and .data sections in VX memory
-- arg	same as -e
-o file	file will be the file of ld directives
-L file	the Low memory file
-H file	the High memory file

Using f77 - Examples

The examples below show some of the posibilites of using f77.

Example 1 Compile and link for the iPSC/2 SRM producing executable host file .

```
f77 host.f sub.f -o host -host
```

Example 2 Compile and link for iPSC/2 SX node producing executable node file .

```
f77 -c -sx -OLM node.f
ld -sx node.o node -lstnode
```

Example 3 Vectorize, compile and link a node program for a VX and SX node

```
f77 node.f sub.v -sx -OLM -lstnode -lvxsxvec
```

Example 4 Vectorize, compile, specify all vector operands to VX memory and link for a VX node

```
f77 node.f sub.v -OLM -e node.o sub.o -node -vec -vx
```

Compiling iPSC/1 Source Programs

Source programs created for the iPSC/1 are not directly compatible with the current iPSC/2 software but may be compiled, linked, and run on the iPSC/2 if the proper software libraries are used and certain precautions are observed.

Host programs must specify a process id to the system before any other calls to the system are made. Therefore, when using the iPSC/1 library, you should call *copen ( )* with a process ID before making any other calls to the library. Subsequent *copen* calls should use the same process ID. (Other process ID's will be ignored.)

To compile and link your C iPSC/1 host program on the iPSC/2:

cc -o *hostexec host1.c host2.c* **... /usr/ipsc/lib/chost.a -host**

To compile and link your iPSC/1 node program:

cc -o *nodeexec node1.c node2.c* **... /usr/ipsc/lib/Llibcnode.a -node**

To compile and link your iPSC/1 FORTRAN host program on the iPSC/2:

f77 -o *hostexec host1.f host2.f* **... /usr/ipsc/lib/fhost.a -host**

To compile and link your iPSC/1 FORTRAN node program on the iPSC/2:

f77 -o *nodeexec node1.f node2.f* **... /usr/ipsc/lib/Llibfnode.a -node**

Using Make with Source Programs

You may compile and link your iPSC/2 programs without using *make*, following the procedures in the previous sections. The *make* command is useful however because it allows you to automatically process a set of files with UNIX tools as you would ordinarily do from the terminal. If the *makefile* is written correctly, *make* also automatically updates any files (via compilation for example) that may need to be changed because of a change in their source files. You may use *make* to compile and link node, host, and associated files for your application.

This section shows the listing of the file *makefile* in */usr/ipsc/examples/c/pi*. You may use a *makefile* similar to this one to compile and link your own host and node programs. For further information see the UNIX System V Programmer's Guide chapter on *make*.

```
# This file is used to compile and link the host2.f, prompt.f
# and node2.f files for the numerical integration example in
# Chapter 4 of the iPSC/2 FORTRAN Programmer's Guide
# The command "make all" causes the compilation and linking

all:     host node

host:    host.f prompt.o
             f77 -o host host.f prompt.o -host
node:    node.f
             f77 -o node node.f -node
```

Figure 4-2
A Possible Makefile for FORTRAN Programs

```
# This file is used to compile and link the host.c, prompt.c
# and node.c files for the numerical integration example in
# Chapter 4 of the iPSC/2 C Programmer's Guide
# the command "make all" causes the compilation and linking

all:     host  node

host:    host.c prompt.o
             cc -o host host.c prompt.o -host
node:    node.c
             cc -o node node.c -node
```

Figure 4-3
A Makefile for C Programs: /usr/ipsc/examples/c/pi/makefile

EXAMPLE C PROGRAM

This section leads you through the compilation, linking and running of an example C program. The source code of the C program is shown in this section.

This example evaluates the definite integral $\int(1/(x^2 + 1))dx$ between 0 and 1. This has the value pi. The numerical integration is done by the n point rectangle quadrature rule. This method requires a choice of the number of points in the calculation which determines the accuracy of the evaluation. The more points chosen the more accurate the answer but the more computation (and time!) is required.

In this example, you enter the number of points for the evaluation and the number of nodes used in the parallel processing. The output to the terminal consists of the integrals' value and the time it took to do the calculation. Through varying the number of points in the calculation and the number of nodes doing the calculation you can get a feel for the effect of multiple processors working simultaneously on the problem. Perform the following steps to run this example:

1. Copy the files in the directory */usr/ipsc/examples/c/pi* to a directory of your choice. You will see this list of files: *host.c*, *node.c*, *prompt.c* and *makefile*.

2. Enter: **make all**
   Examine the *makefile* to learn the specifics of the compilation and linking for this example.

3. This step is <u>not</u> required if you were made a user by means of the sysadm command.

   Set the environment variable TTY with the following commands if your shell is a Bourne shell:

   Enter: **TTY = `tty`**
   export TTY

   OR

   Set the environment variable TTY with the following commands if your shell is a C shell:

   Enter: **setenv TTY `tty`**

   This environment variable is used by several cube commands and must be set before you use the cube. Use the "tty" command to discover your terminal name.

4. Enter: **getcube -c example -t8**

5. Enter: **host**

   The host program is a loop that requests the number of points in the calculation and the number of nodes to do the calculation and then sends this information to the nodes. This program allows you to do the calculation with a set of nodes that is smaller than the number of nodes you have in your cube, the number resulting from the *getcube* command. The cube then does the calculation and returns the answer to the host. (Note: Pi = 3.14159 26535 89793 23846 . . .).

6. When finished, remember to release your cube.

   Enter: **relcube**

Explanation of the Host.c Program

The main body of the host program is shown in part 2 of the listing of the host example program. This host program first loads all the nodes and then goes into a loop. In each cycle of the loop, the user enters the cube dimension and the number of points for the calculation. (The user can also at this point exit the loop and end the program.) The host then sends this information, and the integration parameters, as a message to the nodes. The nodes divide up the integration, do their individual calculations, and send their individual results to node 0 which combines them. Node 0 sends the overall result back as a message to the host. The host prints out the integration result and calculation time, and starts the loop again.

The Host.c Program (part 1):

```
/*
* This program calculates the value of pi, using numerical integration
* with parallal processing, and clocks the solution time.
*
* The user selects the number of processors and the number of points
* of integration. By selecting and timing different cube sizes, you
* obtain a measure of the speedup with perfectly parallel problems.
*/

#define HOST_PID 100      /* process id of the host process */
#define PART_TYPE 10      /* type of partial integration message */
#define APPL_PID   0      /* process id for integ. APPLication */
#define INIT_TYPE  0      /* type of INITial message into cube */
#define SIZE_TYPE  2      /* type of SIZE message */
#define ALL_NODES -1      /* short for all nodes in cube */
#define ALL_PIDS  -1      /* symbol for "all processes */
#define TIME_FACTOR 50.0 /* conversion factor to obtain time */

int size,                 /* size of requested cube */
double  integral;         /* integral value */

struct msg_type {       /* structure for parameters of integration */
        double  a,       /* lower limit of integration */
                b;       /* higher limit of integration */
        long  points     /* number of points in quadrature rule */
};

struct msg_type init; /* integration parameters */

long  tms, ms, tsec, sec, min;    /* for figuring time */
```

The Host.c Program (part 2)

```
main()                  /* begin host */
{

    setpid(HOST_PID);

                        /*load all nodes at pid APPL_PID */
    load ("node", ALL_NODES, APPL_PID);

    for (;;) {          /* begin loop */

                        /* get user input */
        if (!user_input(&msg, &size)) break;

                        /* send sync message of SIZE_TYPE to
                         * ALL_NODES at APPL_PID from size
                         * of length size */
        csend(SIZE_TYPE, &size, sizeof(size), ALL_NODES, APPL_PID);

                        /* send sync message of INIT_TYPE to
                         * ALL_NODES at APPL_PID from init
                         * of length init */
        csend(INIT_TYPE, &msg, sizeof(init), ALL_NODES, APPL_PID);
        crecv(PART_TYPE, &msg, sizeof(integral));
        integral = msg.a;

        tms  = msg.points;
        ms   = tms % 1000;
        tsec = (tms - ms) / 100;
        sec  = tsec % 60;
        min  =  (tsec - sec) / 60;

        printf("\t\tpi is approximately : %.16f\n\n", integral);
        printf(    elapsed time = %d min. %d.%d secs.\n\n",min,sec,ms);

    }                   /* end  loop */

    killcube(ALL_NODES, ALL_PIDS);

}                       /* end main host */
```

Explanation Of The Node.c Program

Part 1 of the listing of the example node program declares various variables and defines some constants. Part 2 shows the main body of the program.

In this program each node first determines its pid and node number and then goes into a loop. For each loop: the node waits to receive the cube dimension entered by the user and finds the total number of nodes to be used in the calculation from this dimension. The node then starts its calculation only if its node number is smaller than the total number of nodes to be used. (It is this condition that limits the nodes working on the calculation to the number of nodes specified.) Next, the integration parameters and total number of points are received from the host. The slice (the width of the rectangle) size is calculated. Each node will calculate the area of many slices, but the number of slices may be different for different nodes.

This difference in slice numbers occurs when the number of nodes used does not divide evenly into the number of slices. Some of the nodes will then have extra slices. The number of points modulus the number of nodes used is the total number of these extra slices.

If the node number is smaller than the number of extra slices then it is given one more slice, otherwise the node uses the number of basic slices.

The lower and upper integration limits for the node is then calculated. The area of the node slices is calculated in a loop covering only the integration limits for the node. Each slice area is the slice width times the function value at the middle of the slice and all areas for the node are added.

A special for loop is used to add the results of all the nodes' calculations. Each node sends its contribution to node 0. Node 0 is the root node for the routine and receives the contribution of all the areas calculated in each node and adds them, then calculates the time taken for the integration. This resulting area is sent as a message to the host and the node loop continues and waits for more values from the host user input.

```
/*
* This program calculates the value of pi, using numerical integration
* with parallal processing, and clocks the solution time.
*
* The user selects the number of processors and the number of poings
* of integration. By selecting and timing different cube sizes you
* obtain a measure of the speedup with perfectly parallel problems.
*
*      Each node:
* 1) receives the integration parameters by invoking crecv,
* 2) calculates a partial sub-interval
* 3) adds its result to the cummulative
*    integral and send the total back to the host (HOST).
*/

#define HOST_PID 100    /* process id of the host process */
#define INIT_TYPE  0    /* type of initialization message */
#define PART_TYPE 10    /* type of partial integration message */
#define SIZE_TYPE  2    /* type of size message */
#define ROOT       0    /* node id of root */

int work_nodes,         /* number of nodes in requested cube */
    my_pid,             /* process id of the node processes */
    my_node;            /* node id returned by mynode() */

long basic_slices,      /* mininum of slices to each node */
    extra_slices,
    my_slices,          /* # of slices to integrate */
    starttime;          /* start time of calculation */

double x,               /* x var. of function to integrate */
    f(),                /* function to integrate */
    slice_size,         /* size of each integration slice */
    partial_int,        /* partial integral */
    sum,
    p2,
    my_a,               /* local lower limit of integration */
    my_b;               /* local higher limit of integration */

   struct msg_type {      /*structure for parameters of integration*/
       double  a,          /* lower limit of integration */
               b;          /* higher limit of integration */
       long  points;       /* number of points in quadrature rule */
   };

   struct msg_type integral;
```

The Node.c Program (part 2)

```
main() {                /* begin main for node program */

  int j;

  my_pid = mypid();     /* get process id */
  my_node = mynode(); /* get node number */

  for (;;) {            /* infinite loop */

                        /* receive number of nodes in work_nodes */
      crecv(SIZE_TYPE, &work_nodes, sizeof(work_nodes));

                        /* if my_node is within the # of nodes */
      if (my_node < work_nodes) {

                        /* receive the total integration parameters */
          crecv(INIT_TYPE, &integral, sizeof(integral));
          starttime = mclock();

                        /* calculate slice size*/
          slice_size   = (integral.b - integral.a) / integral.points;
          basic_slices = integral.points / work_nodes;
          extra_slices = integral.points % work_nodes;

                        /* calc # of slices per node
                         * and local sub-interval */
          if (my_node < extra_slices) {
              my_slices = basic_slices + 1;
              my_a      = integral.a + slice_size * my_node * my_slices;
          }
          else {
              my_slices = basic_slices;
              my_a      = integral.a + slice_size *
                    (my_node * my_slices + extra_slices);
          }
          my_b = my_a + slice_size * my_slices;

                         /*calc. partial integral onsub-interval */
          partial_int = 0.0;
          for (x = my_a; x < my_b - (slice_size * 0.5); x += slice_size) {
               partial_int += f(x + slice_size/2) * slice_size;
          }
```

The Node.c Program (part 3)

```
      /* if we're node 0, gather the contributions from
       * all the other nodes, add them up, and send the
       * result to the host.  also, figure elapsed time
       * and send it in message
       */
      if (mynode() == 0) {
          sum = partial_int;
          for (j = 1; j <work_nodes; j++) {
              crecv(PART_TYPE, &p2, sizeof(p2) );
              sum = sum + p2;
          }
          integral.a = sum;
          integral.points = mclock() - starttime;
          csend(PART_TYPE, &integral,sizeof(integral),myhost(), HOST_PID);
      }
      else csend(PART_TYPE, &partial_int, sizeof(double), 0, mypid() );

    }                   /* end if mynode  < work_nodes */
  }                     /* end of for loop */
}                       /* end main node prog */


/*   function f for integration (pi) */

double f(x)
    double x;
{
    return (4.0/(1.0+x*x));
}
```

The Prompt.c Program

```
/*
*   Function which prompts the user for the integration parameters
*/

struct init_type {      /* structure for parameters of integration */
    double  a,          /* lower limit of integration */
    b;                  /* higher limit of integration */
    long  points;       /* number of points in quadrature rule */
};

user_input (param, size)
    struct init_type *param;
    int *size;
{
    int ans=0;

    printf("\nThis example uses the iPSC to integrate the function:\n");
    printf("                  f(x) = 4 / (1 + x**2)\n");
    printf("between x=0 and x=1.\n");
    printf("The method used is the n-point rectangle quadrature
rule.\n\n");
    printf("How many points do you want (0 or neg. value quits) ? ");
    scanf ("%ld", &param->points);

    if (param->points > 0) {
         while (ans == 0) {
             printf("How many nodes (0-%d) should I use ? ", numnodes());
             scanf ("%d", size);
             printf("\n");
             param->a = 0.0;
             param->b = 1.0;
             if (*size <= numnodes() && *size >= 0)
                 ans = 1;
         }
    }
     else
      ans = 0;

       return ans;
}
```

AN INTRODUCTION TO THE C SHELL

William Joy

AN INTRODUCTION TO THE C SHELL

William Joy
(revised for 4.3BSD by Mark Seiden)

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX™ systems. It incorporates good features of other shells and a history mechanism similar to the redo of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to csh are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with csh is possible after reading just the first section of this document. The second section describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Additional information includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

INTRODUCTION

A shell is a command language interpreter. Csh is the name of one particular command interpreter on UNIX. The primary purpose of csh is to translate command lines typed at a terminal into system actions, such as invocation of other programs. Csh is a user program just like any you might write. Hopefully, csh will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX User Reference Manual. The csh documentation in Appendix A of this manual provides a full description of all features of the shell and is the definitive reference for questions about the shell.

Many words in this document are shown with underlines. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

UNIX is a trademark of Bell Laboratories.

ACKNOWLEDGEMENTS

Numerous people have provided good input about previous versions of csh and aided in its debugging and in the debug ging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution.

AN INTRODUCTION TO THE C SHELL

1. Terminal Usage of the Shell

1.1. The Basic Notion of Commands

A shell in UNIX acts mostly as a medium through which other programs are invoked. While it has a set of builtin functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or words interpreted as a command name followed by arguments. Thus the command

```
mail bill
```

consists of two words. The first word mail names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of directories for a file with the name mail which is expected to contain the mail program.

The rest of the words of the command are given as arguments to the command itself when it is executed. In this case we specified also the argument bill which is interpreted by the mail program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the mail command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
    Bill
EOT
%
```

Here we typed a message to send to bill and ended this message with a ^D which sent an end-of-file to the mail program. (Here and throughout this document, the notation '^x' is to be read 'control-x' and represents the striking of the x key while the control key is held down.) The mail program then echoed the characters `EOT' and transmitted our message. The characters `% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the `% ' prompt the shell was reading command input from our terminal. We typed a complete command `mail bill'. The shell then executed the mail program with argument bill and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ^D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another `% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

1.2. Flag Arguments

A useful notion in UNIX is that of a flag argument. While many arguments to commands specify file names or user names, some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character `-' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current working directory. The option -s is the size option, and

```
ls -s
```

causes ls to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The ls command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to Files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called `now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default standard output for the date command and the date command prints the date on its standard output. The shell lets us redirect the standard output of a command through a notation using the metacharacter `>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the date command such that its standard output is the file `now' rather than the terminal. Thus this command places the current date and time into the file `now'. It is important to know that the date command was unaware that its output was going to a file rather than to the terminal. The shell performed this redirection before the command began executing.

One other thing to note here is that the file `now' need not have existed before the date command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option noclobber exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with `>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a `#' character, this `scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the date command above, we don't really want to save the output forever, so we would more likely do

```
date > #now
```

*Note that if your erase character is a `#', you will have to precede the `#' with a `\'. The fact that the `#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files.

1.4. Metacharacters in the Shell

The shell has a large number of special characters (like `>') which indicate special functions. We say that these notations have syntactic and semantic meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of quotation which allows us to use metacharacters without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via mail, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with `% ' (although we can type our input even before it prompts).

1.5. Input from Files; Pipelines

We learned above how to redirect the standard output of a command to a file. It is also possible to redirect the standard input of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the sort command with standard input, where the command normally reads its input, from the file `data'. We would more likely say

```
sort data
```

letting the sort command open the file `data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its standard input. Since we did not redirect the standard input, it would sort lines as we typed them on the terminal until we typed a ^D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a pipeline. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which ls sorts. We could look at the many options of ls to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the sort command, combining it with ls to get what we want.

---

a ^H, as we demonstrated in section 1.1 how this could be set up.

The -n option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the ls command run with the option -s is to be piped to the command sort run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the -r reverse sort option.

```
ls -s | sort -n -r
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the sort command asking it to sort numerically in reverse order (largest first).

The notation introduced above is called the pipe mechanism. Commands separated by `|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX pathnames consist of a number of components separated by `/'. Each component except the last names a directory in which the next component resides, in effect specifying the path of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory `etc' which is a subdirectory of the root directory `/'. Within this directory the file named is `motd' which stands for `message of the day'. A pathname that begins with a slash is said to be an absolute pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the root). Pathnames which do not begin with `/' are interpreted as starting in the current working directory, which is, by default, your home directory and can be changed dynamically by the cd change directory command. Such pathnames are said to be relative to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and `.'s (periods). In fact, all printing characters except `/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character `.' (period) is not a shell-metacharacter and is often used to separate the extension of a file name from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a base portion of a name (a base portion being that part of the name that is left when a trailing `.' and following characters which are not `.' are stripped off). The file `prog.c' might be the source for a C program, the file `prog.o' the corresponding object file, the file `prog.errs' the errors resulting from a compilation of the program and the file `prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

```
prog.*
```

This expression is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with `prog.'. The character `*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the argument list of the command. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we listed them above. The echo command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by filename expansion of the one input word.

Other notations for filename expansion are also available. The character `?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between `[' and `]'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters around a `-' in this notation to denote a range. Thus

```
chap.[1-5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an argument list) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character `.' at the beginning are treated specially. Neither `*' or `?' or the `[' `]' mechanism will match it. This prevents accidental matching of the filenames `.' and `..' in the working directory which have special meaning to the system, as well as other files such as .cshrc which are not normally visible. We will discuss the special role of the file .cshrc later.

Another filename expansion mechanism gives access to the pathname of the home directory of other users. This notation consists of the character `~' (tilde) followed by another user's login name. For instance the word `~bill' would map to the pathname `/usr/bill' if the home directory for `bill' was `/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a convenient way of accessing the files of other users.

A special case of this notation consists of a `~' alone, e.g. `~/mbox'. This notation is expanded by the shell into the file `mbox' in your home directory, i.e. into `/usr/bill/mbox' for me on Ernie Covax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used cd to change to another directory and have found a file you wish to copy using cp. If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters `{' and `}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character `*'. It will either echo an sorted list of filenames in the current working directory, or print the message `No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, `/', `.' or `-' in an argument word to a command is to enclose it with single quotation characters ', i.e.

```
echo '*'
```

There is one special character `!' which is used by the history mechanism of the shell and which cannot be escaped by placing it within ' characters. It and the character ' itself can be preceded by a single `\' to prevent their special meaning. Thus

```
echo \'\!
```

prints

```
'!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

```
echo \'"*'
```

which prints

```
'*
```

since the first `\' escaped the first `'' and the `*' was enclosed between `'' characters.

1.8. Terminating Commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT signal to the cat command by typing ^C on your terminal.* Since cat does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that cat has terminated and prompts you again with `% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like cat did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the mail program in the first example above was terminated when we typed a ^D which generates an end-of-file from the standard input. The shell also terminates when it gets an end- of-file printing `logout'; UNIX then logs you off the system. Since this means that typing too many ^D's can accidentally log us off, the shell has a mechanism for preventing this. This ignoreeof option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a ^D. This is because it read to the end-of-file of our file `prepared.text' in which we placed a message for `bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the cat command would then have written the text through the pipe to the standard input of the mail command. When the cat command completed it would have terminated, closing down the pipeline and the mail command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing a ^\. This will usually provoke the shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file `core' has been created containing information about the running program's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the "core file" is.

If you want to examine the output of a command without having it move off the screen as the output of the

```
cat /etc/passwd
```

command will, you can use the command

```
more /etc/passwd
```

*On some older Unix systems the DEL or RUBOUT key has the same effect. "stty all" will tell you the INTR key value.

The more program pauses after each complete screenful and types '-- More --' at which point you can hit a space to get another screenful, a return to get another line, a `?' to get some help on other commands, or a `q' to end the more program. (More has been replaced with the less command which is described in Appendix B of this manual.)

You can also use more as a filter, i.e.

```
cat /etc/passwd | more
```

works just like the more simple more command above.

For stopping output of commands not involving more you can use the ^S key to stop the typeout. The typeout will resume when you hit ^Q or any other key, but ^Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ^S and ^Q fast enough to paginate the output nicely, and a program like more is usually used.

1.9. What Now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
sysadm moduser
```

Follow menu choices starting with 'chgshell' to check and change your shell to '/bin/csh'.

2. Details on the Shell for Terminal Users

2.1. Shell Startup and Termination

When you login, the shell is started by the system in your home directory and begins by reading commands from a file .cshrc in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now the default file is adequate.

A login shell, executed after you login to the system, will, after it reads commands from .cshrc, read commands from a file .login also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My .login file looks something like:

```
set ignoreeof
set mail = (/usr/spool/mail/bill)
set time = 15 history = 50
```

This file contains several commands to be executed by UNIX each time I login. The first is a set command which is interpreted directly by the shell. It sets the shell variable ignoreeof which causes the shell to not log me off if I hit ^D. Rather, I use the logout command to log off of the system. By setting the mail variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there.

Next I set the shell variable `time' to `15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable `history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its history list, (described later).

The shell will finish processing my .login file and begin reading commands from the terminal, prompting for each with `% '. When I log off (by giving the logout command) the shell will print `logout' and execute commands from the file `.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the `logout' message the shell is committed to terminating and will take no further input from my terminal.

2.2. Shell Variables

The shell maintains a set of variables. We saw above the variables history and time which had values `10' and `15'. In fact, each shell variable has as value an array of zero or more strings. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name = value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable path. This variable contains a sequence of directory names where the shell searches for commands. The set command with no arguments shows the value of all variables currently defined (we usually say set) in the shell. The default value for path will be shown by set to be

```
% set
logname     bill
argv        ()
history     50
home        /usr/bill
path        (/usr/bill/bin  /bin /usr/bin.)
prompt      %
shell       /bin/csh
status      0
%
```

This output indicates that the variable path points to the current directory '/usr/bill/bin', and then `/bin' and `/usr/bin.' Commands which you may write might be in `.' (usually one of your directories). Commands developed at Bell Laboratories live in `/bin' and `/usr/bin'.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory `.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory `.' on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable home which shows your home directory, the variable ignoreeof which can be set in your .login file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable `ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are set or unset.

Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable `ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables noclobber and mail. The metasyntax

```
> filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your .login file. Then trying to do

```
date > now
```

would cause a diagnostic if `now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of `now'. The `>!' is a special metasyntax indicating that clobbering the file is ok. †

† The space between the `!' and the word `now' is critical here, as `!now' would be an invocation of the history mechanism, and have a totally different effect.

2.3. The Shell's History List

The shell can maintain a "history list" into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure (refer to next page) gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file `bug.c', which we `cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as `!$', meaning the last argument to the previous command. Here the `!' is the history mechanism invocation metacharacter, and the `$' stands for the last argument, by analogy to `$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as `!c', which repeats the last command which started with the letter `c'. If there were other commands starting with `c' done recently we could have said `!cc' or even `!cc:p' which would have printed the last command starting with `cc' without executing it.

After this recompilation, we ran the resulting `a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra `-o bug' telling the compiler to place the resultant binary in the file `bug' rather than `a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the `size' command to see how large the binary program images we have created were, and then an `ls -l' command with the same argument list, denoting the argument list `\!*'. Finally we ran the program `bug' to see that its output is indeed correct.

There are other mechanisms available for repeating commands. The history command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in Appendix A of this Manual.

```
% cat bug.c
main()

{
    printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
    printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\e\en/p
    printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784 + 364 + 1028 = 4176b = 0x1050b
bug: 2784 + 364 + 1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x  1 bill    3932 Dec 19 09:41 a.out
-rwxr-xr-x  1 bill    3932 Dec 19 09:42 bug
% bug
hello
```

2.4. Aliases

The shell has an alias mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as cd which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called `newmail' you wish to use, rather than the standard mail program which is called `mail'. If you place the shell command

```
alias mail newmail
```

in your .cshrc file, the shell will transform an input line of the form

```
mail bill
```

into a call on `newmail'. More generally, suppose we wish the command `ls' to always show sizes of files, that is to always do `-s'. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command syntax `dir' which does an `ls -s'. If we then type

```
dir ~bill
```

then the shell will translate this to

```
ls -s /usr/bill
```

Thus the alias mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an ls command after each change directory cd command. We enclosed the entire alias definition in single quotation marks (') to prevent most substitutions from occurring and the character `;' from being recognized as a metacharacter. The `!' here is escaped with a backslash character (\) to prevent it from being interpreted when the alias command is typed in. The `\!*' here substitutes the entire argument list to the pre-aliasing cd command, without giving an error if there were no arguments. The `;' separating commands is used here to indicate that one command is to be done and then the next.

Similarly the definition

```
alias whois 'grep \!^ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

Warning: The shell reads the .cshrc file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More Redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a "standard error output" which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The `>&' here tells the shell to route both the standard error output and the standard output into `file'. Similarly you can give the command

```
command |& lpr
```

to route both standard and standard error output through the pipe to the line printer. ‡

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file. †

---

‡ A command of the form

```
command >&! file
```

exists, and is used when noclobber is set and file already exists.

† If noclobber is set, then an error will result if file does not exist, otherwise the shell will create file if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when noclobber is set.

2.6. Creating Background and Foreground Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands eparated by semicolons, a single job is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls -s | sort -n | more
mail harold
```

If the metacharacter `&' is typed at the end of the commands, then the job is started as a background job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs "in the background" at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the du program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file `usage' and return immediately with a prompt for the next command without out waiting for du to finish. The du program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard like the INTERRUPT, or QUIT signals.

The kill command terminates a background job immediately. Normally this is done by specifying the process number of the job you want killed. Process numbers can be found with the PS command.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular "working directory." The `change directory' command cd changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The `make directory' command, mkdir, creates a new directory. The pwd (`print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% cd newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory newpaper where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your `home' login directory by doing just

```
cd
```

with no arguments. The name `..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name `..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory `programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

2.8. Useful Built-in Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The alias command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., `ls'.

The echo command prints its arguments. It is often used in "shell scripts" or as an interactive command to see what filename expansions will produce.

The history command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called prompt. By placing a `!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution.
Thus you could type

```
set prompt = '\! % '
```

Note that the `!' character had to be escaped here even within ` " characters.

The logout command can be used to terminate a login shell which has ignoreeof set.

The rehash command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The repeat command can be used to repeat a command several times. Thus to make 5 copies of the file one in the file five you could do

```
repeat 5 cat one >> five
```

The setenv command can be used to set variables in the environment. Thus

```
setenv TERM AT386
```

will set the value of the environment variable TERM to `AT386'. A user program env exists which will print out the environment. It might then show:

```
% env
HOME = /usr/bill
SHELL = /bin/sh
PATH = :/usr/bill/bin:/bin:/usr/bin:/usr/local
TERM = AT386
%
```

The source command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the .cshrc file which you wish to take effect right away.

The time command can be used to cause a command to be timed no matter how much CPU time it takes.

Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2 + 1k 3 + 2io 1pf + 0w
% time wc /etc/rc /usr/bill/rc
      52  178  1347 /etc/rc
      52  178  1347 /usr/bill/rc
     104  356  2694 total
0.1u 0.1s 0:00 13% 3 + 3k 5 + 3io 7pf + 0w
%
```

indicates that the cp command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01). The word count command wc used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage `13%' indicates that over the period when it was active the wc command used an average of 13 percent of the available CPU cycles of the machine.

The unalias and unset commands can be used to remove aliases and variable definitions from the shell, and unsetenv removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the foreach built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the csh manual pages (section1) to become familiar with the other facilities which are available to you.

3. Shell Control Structures and Command Scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called shell scripts. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are not useful for. There is a program called make which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a makefile which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this makefile. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

When using the make program, the shell variable SHELL should be set to the name of the shell it should use to invoke shell script commands.

3.3. Invocation and the argv Variable

A csh command script may be interpreted by saying

```
% csh script ...
```

where script is the name of the file containing a group of csh commands and `...' is replaced by a sequence of arguments. The shell places these arguments in the variable argv and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file `script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a `#' character) then a `/bin/csh' will automatically be invoked to execute `script' when you type

```
script
```

If the file does not begin with a `#' then the standard shell `/bin/sh' will be used to execute it. In the standard shell '/bin/sh', you can only execute scripts written for sh and not scripts written for '/bin/csh'.

3.4. Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as variable substitution is done on these words. Keyed by the character `$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable argv to be echoed to the output of the shell script. It is an error for argv to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to `1' if name is set or to `0' if name is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable name. Thus

```
% set argv = (a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of argv or in the example above `a'. Similarly

```
$argv[$#argv]
```

would give `c', and

```
$argv[1-2]
```

would give `a b'. Other notations useful in shell scripts are

```
$n
```

where n is an integer as a shorthand for

```
$argv[n]
```

the nth parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names.

One minor difference between `$n' and `$argv[n]' should be noted here. The form `$argv[n]' will yield an error if n is not in the range `1-$#argv' while `$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form `n-'; if there are less than n components of the given variable then no words are substituted. A range of the form `m-n' likewise returns an empty vector without giving an error when m exceeds the number of elements of the given variable, provided the subscript n is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations `= =' and `!=' compare strings and the operators `&&' and `||' implement the boolean and/or operations.

The shell also allows file enquiries of the form

```
-? filename
```

where `?' is replace by a number of single characters. For instance the expression primitive

```
-e filename
```

tell whether the file `filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form `{ command }' which returns true, i.e. `1' if the command succeeds exiting normally with exit status 0, or `0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable `$status' examined in the next command. Since `$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see Appendix A of this manual for the shell.

3.6. Sample Shell Script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i (*.c)

    if (! -r ~/backup/$i:t) then
        echo $i:t not in backup... not copied
        continue
    endif

    cmp -s $i ~/backup/$i:tab # to set $status

    if ($status != 0) then
        echo new backup of $i
        cp $i ~/backup/$i:t
    endif
end
```

This script makes use of the foreach command, which causes the shell to execute the commands between the foreach and the matching end for each of the values given between `(' and `)' with the named variable, in this case `i' set to successive values in the list. Within this loop we may use the command break to stop executing the loop and continue to prematurely terminate one iteration and begin the next. After the foreach loop the iteration variable (i in this case) has the value at the last iteration.

The other control construct used here is a statement of the form

```
if ( expression ) then
    command
    ...
endif
```

The placement of the keywords here is not flexible due to the current implementation of the shell.†

†The following two formats are not currently acceptable to the shell:

```
if ( expression )        # Won't work!
then
    command
    ...
endif
```

and

```
if ( expression ) then command endif        # Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
    command
```

Here we have escaped the newline for the sake of appearance. The command must not involve `|', `&' or `;' and must not be another control command. The second form requires the final `\' to immediately precede the end-of-line.

The more general if statements above also admit a sequence of else-if pairs followed by a single else and an endif, e.g.:

```
if ( expression ) then
    commands
else if (expression ) then
    commands
...
else
    commands
endif
```

Another important mechanism used in shell scripts is the `:' modifier. We can use the modifier `:r' here to extract a root of a filename. For example

```
% set i = /mnt/foo.bar
% echo $i $i:r
/mnt/foo.bar /mnt/foo
%
```

shows how the `:r' modifier strips off the trailing `.bar'. Other modifiers will take off the last component of a pathname leaving the head `:h' or all but the last component of a pathname leaving the tail `:t'. These modifiers are fully described in the csh manual pages in Appendix A of this manual. It is also possible to use the command substitution mechanism described in the next major section to perform modifications on strings to then reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the `:' modification mechanism.

Note also that the current implementation of the C shell limits the number of colon modifiers on a $ substitution to 1. Thus:

```
% set i = /a/b/c
% echo $i  $i:h:T
```

Produces

```
% /a/b/c  /a/b:T
```

and does not do what one would expect.

Finally, we note that the character `#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a `#' are discarded by the shell. This character can be quoted using `"' or `\' to place it in an argument word.

3.7. Other Control Structures

The shell also has control structures while and switch similar to those of C. These take the forms

```
while ( expression )
    commands
end
```

and

```
switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

For details see Appendix A for csh. C programmers should note that we use breaksw to exit from a switch while break exits a while or foreach loop. A common mistake to make in csh scripts is to use break rather than breaksw in switches.

Finally, csh allows a goto statement, with labels looking like they do in C, i.e.:

```
loop:
    commands
    goto loop
```

3.8. Supplying Input to Commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

The notation `<< 'EOF' means that the standard input for the ed command is to come from the text in the shell script file up to the next line consisting of exactly 'EOF'. The fact that the `EOF' is enclosed in `' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the `<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form `1,$' in our editor script we needed to insure that this `$' was not variable substituted. We could also have insured this by preceding the `$' here with a `\', i.e.:

```
1,\$s/^[ ]*//
```

but quoting the `EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where label is a label in our program. If an interrupt is received the shell will do a `goto label' and we can remove the temporary files and then do an exit command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status `1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The verbose and echo options and the related -v and -x command line options can be used to help trace the actions of the shell. The -n option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that csh will not execute shell scripts which do not begin with the character `#', that is shell scripts that do not begin with a comment. However, '/bin/sh' will attempt to run scripts that do begin with '#', generally failing.

There is also another quotation mechanism using double quotation marks (") which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as `' does.

4. Other, Less Commonly Used, Shell Features

4.1. Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the foreach control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, `/bin/sh', `/bin/nsh', and `/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use foreach to do this more easily.

```
% foreach i ( 'csh$' 'nsh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with `? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a = (`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The set command here gave the variable a list of all the filenames in the current directory as its value. We can then iterate over these names to perform any chosen function.

The output of a command within an accent or back quote (`) characters is converted by the shell to a list of words. You can also place the ``' quoted string within double quotation (") characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier `:x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in Argument Expansion

Another form of filename expansion, alluded to before involves the characters `{' and `}'. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories `hdrs', `retrofit' and `csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.3. Command Substitution

A command enclosed in accent or back quote (`) characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd = `pwd`
```

to save the current directory in the variable pwd or to do

```
ex `grep -l TRACE *.c`
```

to run the editor ex supplying as arguments those files whose names end in `.c' which have the string `TRACE' in them. Command expansion also occurs in input redirected with `<<' and within `"' quotations. Refer to the csh description in Appendix A for full details.

4.4. Other Details Not Covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions per formed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in the csh description in Appendix A.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the csh description in Appendix A for a list of these options.

APPENDIX A

CSH - A SHELL WITH C-LIKE SYNTAX

csh - a shell (command interpreter) with C-like syntax.

Syntax

csh [-cefinstvVxX] [arg ...]

csh is a first implementation of a command language interpreter. It begins by executing commands from the file '/etc/cshrc' and then '.cshrc' in the home directory of the invoker. If this is a login shell then it also executes commands from the file '.login' there.

In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file '.logout' in the users home directory. If it is present, no error is caused by absence of the files '.login', '.cshrc', or '.logout' in the invoker's home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '&' '|' ';' '<' '>' '(' ')' form separate words. If doubled in '&&', '||', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with '\'. A newline preceded by a '\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations('' or " ") form parts of a word; metachar acters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of single quote (') or double quote (") characters a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by '\' and in quotations using ', `, and ".

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by `|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by `;', and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with `&'.

Any of the above may be placed in `(' `)' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with `||' or `&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See Expressions.)

Substitutions

We now describe the various transformations the shell performs on the input. It performs substitutions in the order the input demands, not necessarily in the order presented here.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character `!' and may begin anywhere in the input stream (with the proviso that they do not nest.) This `!' may be preceded by an `\' to prevent its special meaning; for convenience, a `!' is passed unchanged when it is followed by a blank, tab, newline, `=' or `('. (History substitutions also occur when an input line begins with `^'. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of the history list is controlled by the history variable; the previous command is always retained. Commands are numbered sequentially from 1.

Consider the following output from the history command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an `!' in the prompt string.

With the current event 13 we can refer to previous events by event number `!11', relatively as in `!-2' (referring to the same event), by a prefix of a command word as in `!d' for event 12 or `!wri' for event 9, or by a string contained in a word in the command as in `!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case `!!' refers to the previous command; thus `!!' alone is essentially a redo.

To select words from an event we can follow the event specification by a `:' and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc.
The basic word designators are:

0	first (command) word
n	n'th argument
^	first argument, i.e. `1'
$	last argument
%	word matched by (immediately preceding) ?s? search
x-y	range of words
-y	abbreviates `0-y'
*	abbreviates `^-$', or nothing if only 1 word in event
x*	abbreviates `x-$'
x-	like `x*' but omitting word `$'

The `:' separating the event specification from the word designator can be omitted if the argument selector begins with a `^', `$', `*' `-' or `%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a `:'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing `.xxx' component, leaving the root name.
s/l/r/	Substitute l for r
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. `g&'.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a `g' the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of a slash (/) as the delimiter. A backslash (\) quotes the delimiter into the l and r strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null uses the previous string either from a or from a contextual scan string in '!?s?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing `?' in a contextual scan.

A history reference may be given without an event specification, e.g. '!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus '!?foo?^ !$' gives the first and last arguments from the command matching '?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a caret (^). This is equivalent to '!:s^' providing a convenient shorthand for substitutions on the text of the previous line. Thus '^lb^lib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and `}' if necessary to insulate it from the characters which follow. Thus, after 'ls -ld paul' we might do '!{l}a' to do 'ls -ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by a single quote (') and a double quote (") can be used to prevent all or some of the remaining substitutions. Strings enclosed in ' are prevented any further interpretation. Strings enclosed in " may be expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitition below) does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias Substitution

The shell maintains a list of aliases which can be established, displayed and modified by the alias and unalias commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls -l' the command 'ls /usr' would map to 'ls -l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !^ /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can `alias print "pr \!* | lpr"` to make a command which paginates its arguments to the line printer.

Variable Substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the argv variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the set and unset commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the verbose variable is a toggle which causes command input to be echoed. The setting of this variable results from the -v command line option.

Other operations treat variables numerically. The `@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by `$' characters. This expansion can be prevented by preceding the `$' with a `\' except within `"'s where it always occurs, and within `''s where it never occurs. Strings quoted by ``' are interpreted later (see Command substitution below) so `$' substitution does not occur there until later, if at all. A `$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in `"' or given the `:q' modifier the results of variable substitution may eventually be command and filename substituted. Within `"', a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the `:q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}

Are replaced by the words of the value of variable name, each separated by a blank. Braces insulate name from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter.

If name is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
: May be used to select only some of the words from the value of name. The selector is subjected to `$' substitution and may consist of a single number or two numbers separated by a `-'. The first word of a variables value is numbered `1'. If the first number of a range is omitted it defaults to `1'. If the last member of a range is omitted it defaults to `$#name'. The selector `*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
$#{name}
: Gives the number of words in the variable. This is useful for later use in a `[selector]'.

$0
: Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
: Equivalent to `$argv[number]'.

$*
: Equivalent to `$argv[*]'.

The modifiers `:h', `:t', `:r', `:q' and `:x' may be applied to the substitutions above as may `:gh', `:gt' and `:gr'. If braces `{' '}' appear in the command form, then the modifiers must appear within the braces. Only one `:' modifier on each `$' expansion is allowed.

The following substitutions may not be modified with `:' modifiers.

$?name
${?name}
: Substitutes the string `1' if name is set, `0' if it is not.

$?0
: Substitutes `1' if the current input filename is known, `0' if it is not.

$$
: Substitute the (decimal) process number of the (parent) shell.

Command and Filename Substitution

Command and filename substitution is applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in an accent or back quote (` ). The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within ` "'s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters `*', `?', `[' or `{' or begins with the character `~', then that word is a candidate for filename substitution, also known as `globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters `*', `?' and `[' imply pattern matching, the characters `~' and `{' being more akin to abbreviations.

In matching filenames, the character dot (.) at the beginning of a filename or immediately following a `/', as well as the character `/' must be matched explicitly. The character `*' matches any string of characters, including the null string. The character `?' matches any single character. The sequence `[...]' matches any one of the characters enclosed. Within `[...]', a pair of characters separated by `-' matches any character lexically between the two.

The character `~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. `~' it expands to the invokers home directory as reflected in the value of the variable home. When followed by a name consisting of letters, digits and `-' characters the shell searches for a user with that name and substitutes their home directory; thus `~ken' might expand to `/usr/ken' and `~ken/chmach' to `/usr/ken/chmach'. If the character `~' is followed by a character other than a letter or `/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation `a{b,c,d}e' is a shorthand for `abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus `~source/s1/{oldls,ls}.c' expands to `/usr/source/s1/oldls.c /usr/source/s1/ls.c' whether or not these files exist without any chance of error if the home directory for `source' is `/usr/source'. Similarly `../{memo,*box}' might expand to `../memo ../box ../mbox'. (Note that `memo' was not sorted with the results of matching `*box'.) As a special case `{', `}' and `{}' are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name
: Open file name (which is first variable, command and filename expanded) as the standard input.

<< word
: Read the shell input up to a line which is identical to word. Word is not subjected to variable, filename or command substitution, and each input line is compared to word before any substitutions are done on this input line. Unless a quoting `\', `"', `' or ``' appears in word variable and command substitution is performed on the interven ing lines, allowing `\' to quote `$', `\' and ``'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name
: The file name is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

  If the variable noclobber is set, then the file must not exist or it must be a character special file (e.g. a terminal or `/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the `!' forms can be used and suppress this check.

  The forms involving `&' route the diagnostic output into the specified file as well as the standard output. name is expanded in the same way as `<' input filenames are.

>> name
>>& name
>>! name
>>&! name
: Uses file name as standard output like `>' but places output at the end of the file. If the variable noclobber is set, then it is an error for the file not to exist unless one of the `!' forms is given. Otherwise similar to `>'.

If a command is run detached (followed by &), then the default standard input is the empty file /dev/null. Otherwise, a command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The `<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form `|&' rather than just `|'.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, exit, if, and while commands. The following operators are available:

```
|| && | ^ & = = != = ! <= >= < > << >>
+ - * / % ! ~ ( )
```

Here the precedence increases to the right,

= = and !=
<=, >= <, and >
<< and >>
+ and -
*, /, and %

being, in groups, at the same level. The `= =' and `!=' operators compare their arguments as strings; all others operate on numbers.

Strings which begin with `0' are considered octal numbers. Null or missing arguments are considered `0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser (`&' `|' `<' `>' `(' `)') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in `{' and `}' and file enquiries of the form `-l name' where l is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. `0'. Command executions succeed, returning true, i.e. `1', if the command exits with status 0, otherwise they fail, returning false, i.e. `0'. If more detailed status information is required then the command should be executed outside of an expression and the variable status examined.

Control Flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The foreach, switch, and while statements, as well as the if-then-else form of the if statement require that the major keywords appear in a single simple command on an input line as shown in this section.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Built-In Commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified wordlist as the alias of name; wordlist is command and filename substituted. name is not allowed to be alias or unalias.

break

Causes execution to resume after the end of the nearest enclosing foreach or while. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw

Causes a break from a switch, resuming after the endsw.

case label

A label in a switch statement as discussed below under switch.

cd
cd name

Change the shell's working directory to directory name. If no argument is given then change to the home directory of the user. If this fails but name is a shell variable whose value begins with `/', then this is tried to see if it is a directory.

continue

Continue execution of the nearest enclosing while or foreach. The rest of the commands on the current line are executed.

default:

Labels the default case in a switch statement. The default should come after all case labels.

echo wordlist

The specified words are written to the shells standard output, separated by spaces, and terminated with a newline unless the -n option is specified.

else
end
endif
endsw

See the description of the foreach, if, switch, and while statements below.

exec

The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions.

exit
exit (expr)

The shell exits either with the value of the status variable (first form) or with the value of the specified expr (second form).

foreach name (wordlist)
...
end

The variable name is successively set to each member of wordlist and the sequence of commands between this command and the matching end are executed. (Both foreach and end must appear alone on separate lines.)

The builtin command continue may be used to continue the loop prematurely and the builtin command break to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with `?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist

Like echo but no `\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word

The specified word is filename and command expanded to yield a string of the form `label'. The shell rewinds its input as much as possible and searches for a line of the form `label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history

Displays the history event list.

If (expr) command

If the specified expression evaluates true, then the single command with its arguments is executed. Variable substitution on happens early, at the same time as it does for the rest of the if command. Command must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if expr is false.

if (expr) **then**
...
else if (expr2) **then**
...
else
...
endif

If the specified expr is true then the commands to the first else are executed; otherwise if expr2 is true then the commands to the second else are executed, etc. Any number of else-if pairs are possible; only one endif is needed. The else part is likewise optional. (The words else and endif must appear at the beginning of input lines; the if must appear alone on its input line or after an else.)

kill pid
kill -sig pid...

Sends either the TERM (terminate) signal or the specified signal to the specified process. Signals are given by number.

login

Terminate a login shell, replacing it with an instance of /bin/login. This is one way to log off, included for compatibility with logout. Terminate a login shell. Especially useful if ignoreeof is set.

nice
nice + number
nice command
nice + number command

The first form sets the scheduling priority for this shell to 4. The second form sets the priority to the given number. The final two forms run command at priority 4 and number respectively. The greater the number, the less cpu the process will get. The superuser may specify negative priority by using `nice-number ...'. Command is always executed in a subshell, and the restrictions placed on commands in simple if statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with `&' are effectively nohup'ed.

onintr
onintr-
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form `onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a `goto label' when an interrupt is received or a child process terminates because it was interrupted. In any case, if the shell is running detached and interrupts are being ignored, all forms of onintr have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

rehash

Causes the internal hash table of the contents of the directories in the path variable to be recomputed. This is needed if new commands are added to directories in the path while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count

The specified command which is subject to the same restrictions as the command in the one line if statement above, is executed count times. I/O redirections occur exactly once, even if count is 0.

set
set name
set name = word
set name[index] = word
set name = (wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets name to the null string. The third form sets name to the single word. The fourth form sets the index'th component of name to word; this component must already exist. The final form sets name to the list of words in wordlist. In all cases the value is command and filename expanded.

setenv
setenv name
setenv name value

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs. The first form lists all current environment variables. The last form sets the value of environment variable name to be value, a single string. The second form sets name to an empty string. The most commonly used environment variables TERM and PATH are automatically imported to and exported from the csh variables user, term, and path; there is no need to use setenv for these.

shift
shift variable

The members of argv are shifted to the left, discarding argv[1]. It is an error for argv not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name

The shell reads commands from name. source commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a source at any level terminates all nested source commands. Input during source commands is not placed on the history list.

switch (string)
case string1:
...
breaksw
...
default:
...
breaksw
endsw

Each case label is successively matched, against the specified string which is first command and filename expanded. The file metacharacters `*', `?' and `[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a `default' label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command breaksw causes execution to continue after the endsw. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the endsw.

time

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the time variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
unmask value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by `unalias *'. It is not an error for nothing to be unaliased.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by `unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be unset.

unsetenv pattern

Removes all variables whose name match the specified pattern from the environment. See also the setenv command above.

wait

All background jobs are waited for.

while (expr)
...
end

While the specified expression evaluates non-zero, the commands between the while and the matching end are evaluated. Break and continue may be used to terminate or continue the loop prematurely. (The while and end must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the foreach statement if the input is a terminal.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified name to the value of expr. If the expression contains `<', `>', `&' or `|' then at least this part of the expression must be placed within `(' `)'. The third form assigns the value of expr to the index'th argument of name. Both name and its index'th component must already exist.

The operators `* = ', ` + = ' are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of expr which would otherwise be single words.

Special postfix ` + + ' and `--' operators increment and decrement name respectively, i.e. `@ i + + '.

Pre-defined and Environment Variables

The following variables have special meaning to the shell. Of these, argv, child, home, path, prompt, shell and status are always set by the shell. Except for child and status this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

This shell copies the environment variable PATH into the variable path, TERM into term, and HOME into home, and copies these back into the environment whenever the normal shell variables are reset. It is not necessary to worry about setting path other than in the file .cshrc as inferior csh processes will import the definition of path from the environment, and re-export it if you then change it.

argv — Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. `$1' is replaced by `$argv[1]', etc.

echo — Set when the -x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

history — Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of history may run the shell out of memory. The last executed command is always saved on the history list.

home — The home directory of the invoker, initialized from the environment. The filename expansion of `~' refers to this variable.

ignoreeof — If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.

mail — The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says `You have new mail.' if the file exists with an access time not greater than its modify time.

If the first word of the value of mail is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says `New mail in name' when there is mail in the file name.

noclobber As described in the section on Input/output, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that `>>' redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. `echo [' still gives an error.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no path variable then only full path names will execute. The usual search path is `.', `/bin' and `/usr/bin', but this may vary from system to system. For the super-user the default search path is `/etc', `/bin' and `/usr/bin'. A shell which is given neither the -c nor the -t option will normally hash the contents of the directories in the path variable after reading .cshrc, and each time the path variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the rehash or the commands may not be found.

prompt The string which is printed before each command is read from an interactive terminal input. If a `!' appears in the string it will be replaced by the current event number unless a preceding `\' is given. Default is `% ', or `# ' for the super-user.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of Non-builtin Command Execution below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `1', all other builtin commands set status `0'.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utili zation percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the -v command line option, causes the words of each command to be printed after history substitution.

Non-Builtin Command Execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via exec. Each word in the variable path names a directory from which the shell will attempt to execute the command. If it is given neither a -c nor a -t option, the shell will hash the names in these directories into an internal table so that it will only try an exec in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via or if the shell was given a -c or -t argument, and in any case for each directory component of path which does not begin with a `/', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a sub-shell. Thus `(cd ; pwd) ; pwd' prints the home directory; leaving you where you were (printing this after the home directory), while `cd ; pwd' leaves you in the home directory. Parenthesized commands are most often used to prevent cd from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands and a new shell is spawned to read it.

If there is an alias for shell then the words of the alias will be prepended to the argument list to form the shell command. The first word of the alias should be the full path name of the shell (e.g. `$shell'). Note that this is a special, late occurring, case of alias substitution, and only allows words to be prepended to the argument list without modification.

Argument List Processing

If argument 0 to the shell is `-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in argv.

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file `.cshrc' in the invoker's home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A `\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the verbose variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the echo variable to be set, so that commands are echoed immediately before execution.

-V Causes the verbose variable to be set even before `.cshrc' is executed.

-X Is to -x as -V is to -v.

After processing of flag arguments, if arguments remain but none of the -c, -i, -s, or -t options was given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by `$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a `standard' shell if the first character of a script is not a `#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable argv.

Signal Handling

The shell normally ignores quit signals. Jobs running detached (by `&') are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by onintr. Login shells catch the terminate signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file `.logout'.

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after `.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a `#'.
/tmp/sh*	Temporary file for `<<'.
/etc/passwd	Source of home directories for `~name'.

Limitations

Words can be no longer than 512 characters. The system limits argument lists to 5120 characters. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of alias substitutions on a single line to 20.

Special Characters

The following list summarizes the special characters recognized by **csh**.

Syntactic Metacharacters

- ; Separates commands to be executed sequentially
- | Separates commands in a pipeline
- () Brackets expressions and variable values
- & Follows commands to be executed without waiting for completion

File Name Metacharacters

- / Separates components of a file's path name
- . Separates root parts of a file name from extensions
- ? Expansion character matching any single character
- * Expansion character matching any sequence of characters
- [] Expansion sequence matching any single character from a set of characters
- ~ Used at the beginning of a file name to indicate home directory
- {} Used to specify groups of arguments with common parts

Quotation Metacharacters

- \ Treat following single character as literal
- ' Treat enclosed characters as literal
- ` like ', but allows variable and command expansion

Input/Output Metacharacters

- < Indicates redirected input
- > Indicates redirected output

Expansion/Substitution Metacharacters

- $ Indicates variable substitution
- ! Indicates history substitution
- : Precedes substitution modifiers
- ^ Used in special forms of history substitution
- ` Indicates command substitution

Other Metacharacters

- # Begins scratch file names; indicates C shell comments
- - Prefixes option (flag) arguments to commands

APPENDIX B

LESS - A PROGRAM SIMILAR TO MORE

LESS

NAME

less - opposite of more

SYNOPSIS

less [-dstwcCeEmMqQuU] [-hN] [-b[fp]N] [-xN] [-[z]N]
[-P[mM]string] [-[lL]logfile] [+ cmd] [filename]...

DESCRIPTION

Less is a program similar to more (1), but which allows backwards movement in the file as well as forward movement. Also, less does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like vi (1). Less uses termcap, so it can run on a variety of terminals. There is even limited support for hardcopy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with an up-arrow.)

Commands are based on both more and vi. Commands may be preceeded by a decimal number, called N in the descriptions below. The number is used by some commands, as indicated.

COPYRIGHT

In the following descriptions, ^X means control-X.

H	Help: display a summary of these commands. If you forget all the other commands, remember this one.
SPACE	Scroll forward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed.
f or ^F	Same as SPACE.
b or ^B	Scroll backward N lines, default one window (see option -z below). If N is more than the screen size, only the final screenful is displayed.
RETURN	Scroll forward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size.
e or ^E	Same as RETURN.
j or ^J	Also the same as RETURN.
y or ^Y	Scroll backward N lines, default 1. The entire N lines are displayed, even if N is more than the screen size.
k or ^K	Same as y.
d or ^D	Scroll forward N lines, default 10. If N is specified, it becomes the new default for subsequent d and u commands.
u or ^U	Scroll backward N lines, default 10. If N is specified, it becomes the new default for subsequent d and u commands.
r or ^R or ^L	Repaint the screen.
R	Repaint the screen, discarding any buffered input. Useful if the file is changing while it is being viewed.
g	Go to line N in the file, default 1 (beginning of file). (Warning: this may be slow if N is large.)
G	Go to line N in the file, default the end of the file. (Warning: this may be slow if standard input, rather than a file, is being read.)
p	Go to a position N percent into the file. N should be between 0 and 100. (This is possible if standard input is being read, but only if less has already read to the end of the file. It is always fast, but not always useful.)
%	Same as p.

m	Followed by any lowercase letter, marks the current position with that letter.
'	(Single quote.) Followed by any lowercase letter, returns to the position which was previously marked with that letter. Followed by another single quote, returns to the postion at which the last "large" movement command was executed. All marks are lost when a new file is examined.
/pattern	Search forward in the file for the N-th line containing the pattern. N defaults to 1. The pattern is a regular expression, as recognized by ed. The search starts at the second line displayed (but see the -t option, which changes this).
?pattern	Search backward in the file for the N-th line containing the pattern. The search starts at the line immediately before the top line displayed.
n	Repeat previous search, for N-th line containing the last pattern.
E	Examine a new file. If the filename is missing, the "current" file (see the N and P commands below) from the list of files in the command line is re-examined. If the filename is a pound sign (#), the previously examined file is re-examined.
N	Examine the next file (from the list of files given in the command line). If a number N is specified (not to be confused with the command N), the N-th next file is examined.
P	Examine the previous file. If a number N is specified, the N-th previous file is examined.
= or ^G	Prints some information about the file being viewed, including its name and the byte offset of the bottom line being displayed. If possible, it also prints the length of the file and the percent of the file above the last displayed line.
-	Followed by one of the command line option letters (see below), this will toggle the setting of that option and print a message describing the new setting.
+ cmd	Causes the specified cmd to be executed each time a new file is examined. For example, + G causes less to initially display each file starting at the end rather than the beginning.
V	Prints the version number of less being run.
q	Exits less.

The following two commands may or may not be valid, depending on your particular installation.

v — Invokes an editor to edit the current file being viewed. The editor is taken from the environment variable EDITOR, or defaults to "vi".

! shell-command — Invokes a shell to run the shell-command given. A percent sign in the command is replaced by the name of the current file. "!!" repeats the last shell command.

Command line options are described below. Most options may be changed while less is running, via the "-" command.

Options are also taken from the environment variable "LESS". For example, if you like more-style prompting, to avoid typing "less -m ..." each time less is invoked, you might tell csh:

```
setenv LESS m
```

or if you use sh:

```
LESS = m; export LESS
```

The environment variable is parsed before the command line, so command line options override the LESS environment variable. A dollar sign ($) may be used to signal the end of an option string. This is important only for options like -P which take a following string.

-b — The -bn option tells less to use a non-standard buffer size. There are two standard (default) buffer sizes, one is used when a file is being read and the other when a pipe (standard input) is being read. The current defaults are 5 buffers for files and 12 for pipes. (Buffers are 1024 bytes.) The number n specifies a different number of buffers to use. The -b may be followed by "f", in which case only the file default is changed, or by "p" in which case only the pipe default is changed. Otherwise, both are changed.

-c — Normally, less will repaint the screen by scrolling from the bottom of the screen. If the -c option is set, when less needs to change the entire display, it will paint from the top line down.

-C — The -C option is like -c, but the screen is cleared before it is repainted.

-d — Normally, less will complain if the terminal is dumb; that is, lacks some important capability, such as the ability to clear the screen or scroll backwards. The -d option suppresses this complaint (but does not otherwise change the behavior of the program on a dumb terminal).

-e — Normally the only way to exit less is via the "q" command. The -e option tells less to automatically exit the second time it reaches end-of-file.

-E — The -E flag causes less to exit the first time it reaches end-of-file.

-h — Normally, less will scroll backwards when backwards movement is necessary. The -h option specifies a maximum number of lines to scroll backwards. If it is necessary to move backwards more than this many lines, the screen is repainted in a forward direction. (If the terminal does not have the ability to scroll backwards, -h0 is implied.)

-l

The -l option, followed immediately by a filename, will cause less to copy its input to the named file as it is being viewed. This applies only when the input file is a pipe, not an ordinary file. If the file already exists, less will ask for confirmation before overwriting it.

-L

The -L option is like -l, but it will overwrite an existing file without asking for confirmation.

-m

Normally, less prompts with a colon. The -m option causes less to prompt verbosely (like more), with the percent into the file.

-M

The -M option causes less to prompt even more verbosely than more.

-P

The -P option provides a way to tailor the three prompt styles to your own preference. You would normally put this option in your LESS environment variable, rather than type it in with each less command. Such an option must either be the last option in the LESS variable, or be terminated by a dollar sign. -P followed by a string changes the default (short) prompt to that string. -Pm changes the medium (-m) prompt to the string, and -PM changes the long (-M) prompt. The string consists of a sequence of letters which are replaced with certain predefined strings, as follows:

F	file name
f	file name, only once
O	file n of n
o	file n of n, only once
b	byte offset
p	percent into file
P	percent if known, else byte offset

Angle brackets, < and >, may be used to surround a literal string to be included in the prompt. The defaults are "fo" for the short prompt, "foP" for the medium prompt, and "Fobp" for the long prompt.

Example: Setting your LESS variable to "PmFOP$PMFObp" would change the medium and long prompts to always include the file name and "file n of n" message.

Another example: Setting your LESS variable to "mPm<--Less-->FoPe" would change the medium prompt to the string "--Less--" followed by the file name and percent into the file. It also selects the medium prompt as the default prompt (because of the first "m").

-q

Normally, if an attempt is made to scroll past the end of the file or before the beginning of the file, the terminal bell is rung to indicate this fact. The -q option tells less not to ring the bell at such times. If the terminal has a "visual bell", it is used instead.

-Q

Even if -q is given, less will ring the bell on certain other errors, such as typing an invalid character. The -Q option tells less to be quiet all the time; that is, never ring the terminal bell. If the terminal has a "visual bell", it is used instead.

-s The -s option causes consecutive blank lines to be squeezed into a single blank line. This is useful when viewing nroff output.

-t Normally, forward searches start just after the top displayed line (that is, at the second displayed line). Thus forward searches include the currently displayed screen. The -t option causes forward searches to start just after the bottom line displayed, thus skipping the currently displayed screen.

-u If the -u option is given, backspaces are treated as printable characters; that is, they are sent to the terminal when they appear in the input.

-U If the -U option is given, backspaces are printed as the two character sequence "^H".

If neither -u nor -U is given, backspaces which appear adjacent to an underscore character are treated specially: the underlined text is displayed using the terminal's hardware underlining capability. Also, backspaces which appear between two identical characters are treated specially: the overstruck text is printed using the terminal's hardware boldface capability. Other backspaces are deleted, along with the preceeding character.

-w Normally, less uses a tilde character to represent lines past the end of the file. The -w option causes blank lines to be used instead..

-x The -xn option sets tab stops every n positions. The default for n is 8.

-[z] When given a backwards or forwards window command, less will by default scroll backwards or forwards one screenful of lines. The -zn option changes the default scrolling window size to n lines. If n is greater than the screen size, the scrolling window size will be set to one screenful. Note that the "z" is optional for compatibility with more.

+ If a command line option begins with +, the remainder of that option is taken to be an initial command to less. For example, +G tells less to start at the end of the file rather than the beginning, and +/xyz tells it to start at the first occurence of "xyz" in the file. As a special case, +<number> acts like +<number>g; that is, it starts the display at the specified line number (however, see the caveat under the "g" command above). If the option starts with ++, the initial command applies to every file being viewed, not just the first one. The + command described previously may also be used to set (or change) an initial command for every file.

When used on standard input (rather than a file), you can move backwards only a finite amount, corresponding to that portion of the file which is still buffered. The -b option may be used to expand the buffer space.

APPENDIX C

STRINGS

STRINGS

NAME

strings - find printable strings in files

SYNOPSIS

strings [-a] [-n #] [-o] ... file ...

DESCRIPTION

Strings displays ascii strings contained in the named input files. The input files may be either ordinary files or COFF object files. For ordinary files, the entire file is searched for strings. COFF files have only their .data sections searched unless the -a option is used which also searches any .text sections.

The strings extracted by strings are defined to be a sequence of at least 4 (by default) printable characters terminated by either a NULL or a NEWLINE. The -n # option changes the minimum number of characters. From 1 to BUFSIZ characters are possible.

The -o switch causes strings to prefix each output line with the octal offset within the file (which is probably not the COFF memory address).

IN MEMORIUM

Strings has been modeled after the Berkeley strings(1) tool. It is not a Berkeley product.

COPYRIGHT

AUTHOR

Tom Reynolds
Phoenix microsystems, inc.
991 Discovery Drive
Huntsville, AL 35806

APPENDIX D

TERMS

TERMS

This section lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form `pr (1)' indicate that the command pr is in the UNIX User Reference manual in section 1.

References of the form (2.5) indicate that more information can be found in section 2.5 of this appendix (An Introduction to the C Shell).

- Your current directory has the name `.' as well as the name printed by the command pwd; see also dirs. The current directory `.' is usually the first component of the search path contained in the variable path, thus commands which are in `.' are found first (2.2). The character `.' is also used in separating components of filenames (1.6). The character `.' at the beginning of a component of a pathname is treated specially and not matched by the filename expansion metacharacters `?', `*', and `[' `]' pairs (1.6).

- Each directory has a file `..' in it which is a reference to its parent directory. After changing into the directory with chdir, i.e.

  chdir paper

  you can return to the parent directory by doing

  chdir ..

  The current directory is printed by pwd (2.7).

a.out

Compilers which create executable images create them, by default, in the file a.out. for historical reasons (2.3).

absolute pathname

A pathname which begins with a `/' is absolute since it specifies the path of directories from the beginning of the entire directory system - called the root directory. Pathnames which are not absolute are called relative (see definition of relative pathname) (1.6).

alias

An alias specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command alias which establishes aliases and can print their current values. The command unalias is used to remove aliases (2.4).

argument

Commands in UNIX receive a list of argument words. Thus the command

```
echo a b c
```

consists of the command name `echo' and three argument words `a', `b' and `c'. The set of arguments after the command name is said to be the argument list of the command (1.1).

argv

The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called argv within the shell. This name is taken from the conventional name in the C programming language (3.4).

background

Commands started without waiting for them to complete are called background commands (2.6).

base

A filename is sometimes thought of as consisting of a base part, before any `.' character, and an extension - the part after the `.'. See filename and extension (1.6) and basename (1).

bin

A directory containing binaries of programs and shell scripts to be executed is typically called a bin directory. The standard system bin directories are `/bin' containing the most heavily used commands and `/usr/bin' which contains most other user programs. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a component of the variable path.

break

Break is a builtin command used to exit from loops within the control structure of the shell (3.7).

breaksw The breaksw builtin command is used to exit from a switch control structure, like a break exits from loops (3.7).

builtin A command executed directly by the shell is called a builtin command. Most commands in UNIX are not built into the shell, but rather exist as files in bin directories. These commands are accessible because the directories in which they reside are named in the path variable.

case A case command is used as a label in a switch statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation `csh (1)' (3.7).

cat The cat program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal, to `cat a file' (1.8, 2.3).

cd The cd command is used to change the working directory. With no arguments, cd changes your working directory to be your home directory (2.4, 2.7).

cmp Cmp is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program diff, described in `diff (1)' is used.

command A function performed by the system, either by the shell (a builtin command) or by a program residing in a file in a directory within the UNIX system, is called a command (1.1).

command name When a command is issued, it consists of a command name, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution The replacement of a command enclosed in accent or back quotes (`) characters by the text output by that command is called command substitution (4.3).

component A part of a pathname between `/' characters is called a component of that pathname. A variable which has multiple strings as value is said to have several components; each string is a component of the variable.

continue A builtin command which causes execution of the enclosing foreach or while loop to cycle prematurely. Similar to the continue command in the programming language C (3.6).

control- Certain special characters, called control characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus control-c is produced by holding down the CONTROL key while pressing the `c' key. Usually UNIX prints an caret (^) followed by the corresponding letter when you type a control character (e.g. `^C' for control-c (1.8).

core dump When a program terminates abnormally, the system places an image of its current state in a file named `core'. This core dump can be examined with the system debugger `sdb (1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form

Illegal instruction (core dumped)

(where `Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the `core' file.

cp The cp (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).

csh The name of the shell program that this document describes.

.cshrc The file .cshrc in your home directory is read by each shell as it begins execution. It is usually used to change the setting of the variable path and to set alias parameters which are to take effect globally (2.1).

date The date command prints the current date and time (1.3).

debugging Debugging is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging (4.4).

default: The label default: is used within shell switch statements, as it is in the C language to label the code to be executed if none of the case labels matches the value switched on (3.7).

DELETE The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ^C.

detached A command that continues running in the background after you logout is said to be detached.

diagnostic An error message produced by a program is often referred to as a diagnostic. Most error messages are not written to the standard output, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the diagnostic output which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal (2.5).

directory A structure which contains files. At any time you are in one particular directory whose names can be printed by the command pwd. The chdir command will change you to another directory, and make the files in that directory visible. The directory in which you are when you first login is your home directory (1.1, 2.7).

du The du command is a program (described in `du (1)') which prints the number of disk blocks is all directories below and including your current working directory (2.6).

echo The echo command prints its arguments (1.6, 3.6).

else The else command is part of the `if-then-else-endif' control command construct (3.6).

endif If an if statement is ended with the word then, all lines following the if up to a line starting with the word endif or else are executed if the condition between parentheses after the if is true (3.6).

EOF An end-of-file is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a pipe receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

escape A character `\' used to prevent the special meaning of a metacharacter is said to escape the character from its special meaning. Thus

```
echo \*
```

will echo the character `*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ escapes `*' (1.7). There is also a non-printing character called escape, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be suspended. Most systems use control-s to stop the output and control-q to start it.

/etc/passwd

This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by `:' characters (1.8). You can look at this file by saying

```
cat /etc/passwd
```

The commands finger and grep are often used to search for information in this file. See `passwd(5)', and `grep (1)' for more details.

exit

The exit command is used to force termination of a shell script, and is built into the shell (3.9).

exit status

A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its exit status, a status of zero being considered `normal termination'. The exit command can be used to force a shell command script to give a non-zero exit status (3.6).

expansion

The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of expansion. Thus the replacement of the word `*' by a sorted list of files in the current directory is a `filename expansion'. Similarly the replacement of the characters `!!' by the text of the last command is a `history expansion'. Expansions are also referred to as substitutions (1.6, 3.4, 4.2).

expressions

Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell expressions are those of the language C (3.5).

extension

Filenames often consist of a base name and an extension separated by the character `.'. By convention, groups of related files often share the same root name. Thus if `prog.c' were a C program, then the object file for this program would be stored in `prog.o'.

filename

Each file in UNIX has a name consisting of up to 14 characters and not including the character `/' which is used in pathname building. Most filenames do not begin with the character `.', and contain only letters and digits with perhaps a `.' separating the base portion of the filename from an extension (1.6).

filename expansion

Filename expansion uses the metacharacters `*', `?' and `[' and `]' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter `~' and allow files in other users' directories to be named easily (1.6, 4.2).

flag

Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as flag options, and by convention consist of one or more letters preceded by the character `-' (1.2). Thus the ls (list files) command has an option `-s' to list the sizes of files. This is specified

```
ls -s
```

foreach

The foreach command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

goto

The shell has a command goto used in shell scripts to transfer control to a given label (3.7).

grep

The grep command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file /etc/passwd which contains the string `bill'. Actually, grep scans for regular expressions in the sense of the editors `ed (1)' and `ex (1)'. Grep stands for `globally find regular expression and print' (2.4).

history

The history mechanism of the shell allowsprevious commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a history list where these commands are kept, and a history variable which controls how large this list is (2.3).

home directory

Each user has a home directory, which is given in your entry in the password file, /etc/passwd. This is the directory which you are placed in when you first login. The cd or chdir command with no arguments takes you back to this directory, whose name is recorded in the shell variable home. You can also access the home directories of other users in forming filenames using a filename expansion notation and the character `~' (1.6).

if

A conditional command within the shell, the if command is used in shell command scripts to make decisions about what course of action to take next (3.6).

ignoreeof

Normally, your shell will exit, printing `logout' if you type a control-d at a prompt of `% '. This is the way you usually log off the system. You can set the ignoreeof variable if you wish in your .login file and then use the command logout to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).

input

Many commands on UNIX take information from the terminal or from files which they then act on. This information is called input. Commands normally read for input from their standard input which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character `<'. Many commands will also read from a file specified as argument. Commands placed in pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a filename to use as standard input. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).

interrupt

An interrupt is a signal to a program that is generated by typing ^C. (On older versions of UNIX the RUBOUT or DELETE key were used for this purpose.) It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt because many commands die when they receive an interrupt (1.8, 3.9).

kill

A command which sends a signal to a job causing it to terminate (2.6).

.login

The file .login in your home directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially set commands to the shell itself (2.1).

login shell

The shell that is started on your terminal when you login is called your login shell. It is different from other shells which you may run (e.g. on shell scripts) in that it reads the .login file before reading commands from the terminal and it reads the .logout file after you logout (2.1).

logout

The logout command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an end-of-file, but if you have set ignoreeof in you .login file then this will not work and you must use logout to log off the UNIX system (2.8).

.logout

When you log off of UNIX the shell will execute commands from the file .logout in your home directory after it prints `logout'.

lpr

The command lpr is the line printer daemon. The standard input of lpr spooled and printed on the UNIX line printer. You can also give lpr a list of filenames as arguments to be printed. It is most common to use lpr as the last component of a pipeline (2.3).

ls

The ls (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful flag arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail

The mail program is used to send and receive messages from other UNIX users (1.1, 2.1), whether they are logged on or not.

make

The make command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).

makefile

The file containing commands for make is called *makefile* or *Makefile* (3.2).

manual

The *manual* often referred to is the `UNIX manual'. It contains 8 numbered sections with a description of each UNIX program (section 1), system call (section 2), subroutine (section 3), device (section 4), special data structure (section 5), game (section 6), miscellaneous item (section 7) and system administration program (section 8). There are also supplementary documents (tutorials and reference guides) for individual programs which require explanation in more detail.

metacharacter

Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called metacharacters. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be quoted. An example of a metacharacter is the character `>' which is used to indicate placement of output into a file. For the purposes of the history mechanism, most unquoted metacharacters form separate words (1.4). The appendix to this user's manual lists the metacharacters in groups by their function.

mkdir

The mkdir command is used to create a new directory.

modifier

Substitutions with the history mechanism, keyed by the character `!' or of variables using the metacharacter `$', are often subjected to modifications, indicated by placing the character `:' after the substitution and following this with the modifier itself. The command substitution mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

more

The program more writes a file on your termi nal allowing you to control how much text is displayed at a time. More can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

noclobber

The shell has a variable noclobber which may be set in the file .login to prevent accidental destruction of files by the `>' output redirection metasyntax of the shell (2.2, 2.5).

noglob

The shell variable noglob is set to suppress the filename expansion of arguments containing the metacharacters `~', `*', `?', `[' and `]' (3.6).

onintr

The onintr command is built into the shell and is used to control the action of a shell command script when an interrupt signal is received (3.9).

output

Many commands in UNIX result in some lines of text which are called their output. This output is usually placed on what is known as the standard output which is normally con nected to the user's terminal. The shell has a syntax using the metacharacter `>' for redirecting the standard output of a command to a file (1.3). Using the pipe mechanism and the metacharacter `|' it is also possible for the standard output of one command to become the standard input of another command (1.5). Certain commands such as the line printer daemon p do not place their results on the standard output but rather in more useful places such as on the line printer (2.3). Similarly the write command places its output on another user's terminal rather than its standard output (2.3). Commands also have a diagnostic output where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special metanotation (2.5).

path

The shell has a variable path which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the path variable, left to right. Since the normal definition of the path variable is

```
path (. /bin /usr/bin)
```

the shell normally looks in the current directory, and then in the standard system directories `/bin' and `/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have `execute' permission set. This is normally true because a command of the form

```
chmod 755 script
```

was executed to turn this execute permission on (3.3). If you add new commands to a directory in the path, you should issue the command rehash (2.2).

pathname

A list of names, separated by `/' characters, forms a pathname. Each component, between successive `/' characters, names a directory in which the next component file resides. Pathnames which begin with the character `/' are interpreted relative to the root directory in the file system. Other pathnames are interpreted relative to the current directory as reported by pwd. The last component of a pathname may name a directory, but usually names a file.

pipeline

A group of commands which are connected together, the standard output of each connected to the standard input of the next, is called a pipeline. The pipe mechanism used to connect these commands is indicated by the shell metacharacter `|' (1.5, 2.3).

port

The part of a computer system to which each terminal is connected is called a port. Usually the system has a fixed number of ports, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr

The pr command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

env

The env command is used to print the current setting of variables in the environment (2.8).

process

An instance of a running program is called a process (2.6). UNIX assigns each process a unique number when it is started - called the process number. Process numbers can be used to stop individual processes using the kill command when the processes are part of a detached background job.

program

Usually synonymous with command; a binary file or shell command script which performs a useful function is often called a program.

prompt

Many programs will print a prompt on the terminal when they expect input. Thus the editor `ex (1)' will print a `:' when it expects input. The shell prompts for input with `% ' and occasionally with `? ' when reading commands from the terminal (1.1). The shell has a variable prompt which may be set to a different value to change the shell's main prompt. This is mostly used when debugging the shell (2.8).

ps

The ps command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the csh you use to run the ps command, are not normally shown in the output.

pwd

The pwd command prints the full pathname of the current working directory.

quit

The quit signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).

quotation

The process by which metacharacters are prevented their special meaning, usually by using the quote (') character in pairs, or by using the character `\', is referred to as quotation (1.7).

redirection

The routing of input or output from or to a file is known as redirection of input or output (1.3).

rehash

The rehash command tells the shell to rebuild its internal table of which commands are found in which directories in your path. This is necessary when a new program is installed in one of these directories (2.8).

relative pathname

A pathname which does not begin with a `/' is called a relative pathname since it is interpreted relative to the current working directory. The first component of such a pathname refers to some file or directory in the working directory, and subsequent components between `/' characters refer to directories below the working directory. Pathnames that are not relative are called absolute pathnames (1.6).

repeat

The repeat command iterates another command a specified number of times.

root

The directory that is at the top of the entire directory structure is called the root directory since it is the `root' of the entire tree structure of directories. The name used in pathnames to indicate the root is `/'. Pathnames starting with `/' are said to be absolute since they start at the root directory. Root is also used as the part of a pathname that is left after removing the extension. See filename for a further explanation (1.6).

RUBOUT

The RUBOUT or DELETE key is often used to erase the previously typed character; some users prefer the BACKSPACE for this purpose. On older versions of UNIX this key served as the INTR character.

script Sequences of shell commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).

set The builtin set command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the set command the behavior of the shell can be affected (2.1).

setenv Variables in the environment `environ (5)' can be changed by using the setenv builtin command (2.8). The env command can be used to print the value of the variables in the environment.

shell A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called csh.

shell script See script (3.3, 3.10).

signal A signal in UNIX is a short message that is sent to a running program which causes something to happen to that process. Signals are sent either by typing special control characters on the keyboard or by using the kill command (1.8, 2.6).

sort The sort program sorts a sequence of lines in ways that can be controlled by argument flags (1.5).

source The source command causes the shell to read commands from a specified file. It is most useful for reading files such as .cshrc after changing them (2.8).

special character See metacharacters and the appendix to this manual.

standard We refer often to the standard input and standard output of commands. See input and output (1.3, 3.8).

status A command normally returns a status when it finishes. By convention a status of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable status is set to the status returned by the last command. It is most useful in shell commmand scripts (3.6).

string A sequential group of characters taken together is called a string. Strings can contain any printable characters (2.2).

stty The stty program changes certain parameters inside UNIX which determine how your terminal is handled. See `stty (1)' for a complete description (2.6).

substitution The shell implements a number of substitutions where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter `!' and variable substitution indicated by `$'. We also refer to substitutions as expansions (3.4).

switch The switch command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the switch statement in the language C (3.7).

termination When a command which is being executed finishes we say it undergoes termination or terminates. Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an interrupt or quit signal (1.8). The kill program terminates specified jobs (2.6).

then The then command is part of the shell's `if-then-else-endif' control construct used in command scripts (3.6).

time The time command can be used to measure the amount of CPU and real time consumed by a specified command (2.1, 2.8).

tty The word tty is a historical abbreviation for `teletype' which is frequently used in UNIX to indicate the port to which a given terminal is connected. The tty command will print the name of the tty or port to which your terminal is presently connected.

unalias The unalias command removes aliases (2.8).

UNIX UNIX is an operating system on which csh runs. UNIX provides facilities which allow csh to invoke other programs such as editors and text formatters which you may wish to use.

unset The unset command removes the definitions of shell variables (2.2, 2.8).

variable expansion

See variables and expansion (2.2, 3.4).

variables

Variables in csh hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See path, noclobber, and ignoreeof for examples. Variables such as argv are also used in writing shell programs (shell command scripts) (2.2).

verbose

The verbose shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The verbose variable is set by the shell's -v command line option (3.10).

wc

The wc program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).

while

The while builtin control construct is used in shell command scripts (3.7).

word

A sequence of characters which forms an argument to a command is called a word. Many characters which are neither letters, digits, `-', `.' nor `/' form words all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a word by surrounding it with `'' characters except for the characters `'' and `!' which require special treatment (1.1). This process of placing special characters in words without their special meaning is called quoting.

working directory

At any given time you are in one particular directory, called your working directory. This directory's name is printed by the pwd command and the files listed by ls are the ones in this directory. You can change working directories using chdir.

write

The write command is an obsolete way of communicating with other users who are logged in to UNIX (you have to take turns typing).