The Alpha Operating System: Programming Utilities

J. Duane Northcutt
Samuel E. Shipman

TR #88041

Archons Project Technical Report

April 8, 1988

The Alpha Operating System: Programming Utilities

J. Duane Northcutt
Samuel E. Shipman

April 8, 1988

Archons Project Technical Report #88041

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

List of Figures

Abstract

This report describes the programming tools developed for the Alpha operating system effort [Northcutt 87] and their user interfaces. The UNIX-based Sun Microsystems workstations chosen as the development platforms for this effort provided a wide range of valuable programming utilities, but it was nonetheless necessary to construct additional programming support tools, in addition to modifying some of the existing UNIX tools.

Most of the tools described here are not Alpha-specific, but are quite general-purpose in their usefulness and could be used in a number of different applications. However, some of the material covered in this report assumes an understanding of the basic structure of the Alpha testbed, which is described in a companion document [Northcutt 88b].

The tools described in this document allow the user to interact easily and directly with the Alpha testbed. These utilities were designed to abstract away many of the low level hardware details of the testbed, and allow the users to perform Alpha operating system and application code development without requiring a very detailed knowledge of the underlying testbed hardware.

The programming tools described in this document include: the object language pre-processor, which, through a set of simple extensions to the C programming language, provides the programmer with an improved interface to the services provided by the Alpha kernel; the initial configuration generation program, which (via a command set that is compatible with the object language extensions) provides the programmer-in-the-large with a means of specifying the initial configurations of distributed applications (i.e., the initial threads and objects, and their assignment to testbed nodes); the file dependency maintenance program, which assists the programmer in managing the collection of files that make up the Alpha system (as well as applications), by automatically maintaining records of the dependency relationships among files (for use with the UNIX *make* program) and the commands needed to create complete, executable programs; and a series of programs with which the standard UNIX representation of executable files can be manipulated. In addition, this report describes the tools used to up- and down-load programs to/from the UNIX development host machines, and to perform interactive I/O with the testbed machines. The interactive command interface to the monitor program (which resides within the PROMs of each processor in the testbed) is also described in this document, as well as the interface to the testbed's remote power control unit. Finally, the system monitoring and debugging tools that have been used in the development of Alpha and its applications are described.

1 Introduction

This report describes the collection of software tools, used in conjunction with the Alpha Distributed Computer System Testbed [Northcutt 88b], to develop applications and system software on top of the Alpha operating system's kernel interface [Northcutt 88a]. The Alpha programming environment is based on a collection of Sun Microsystems workstations, running SunOS, and provides cross development support for the Alpha testbed. The programming support for Alpha combines standard UNIX software development facilities and custom (Alpha-specific and general-purpose) tools running under UNIX and on the Alpha testbed. These tools provide the client with a means of programming the Alpha system, permitting the creation, execution, monitoring, and debugging of application programs on Alpha.

In addition to the tools described in this report, a great deal of programming effort was expended in order to integrate the Sun Microsystems workstations into the CMU Computer Science Department's computing environment, and to make the Suns a more hospitable system development host. For example, utilities were created or adapted to allow programmers to access the departmental Xerox laser printer (i.e., the *Dover*), permit network interface to the DARPA Internet, allow modem dial-in/-out access to the project's workstations, support document production and editing (i.e., with *Scribe* and *Emacs*), as well as to provide support for other functions vital to the programming environment (e.g., *NetNews*, *BBoards*, and *Coketool*). Because the Alpha project was the first (and for many years, the only) project that used Sun workstations, there was no departmental support (either hardware or software), and so the members of the Alpha team were responsible for providing all forms of software (and hardware) support for the project. Despite the significant amount of effort that went into these support efforts, the results represent enhancements to the workstations' general programming environment (and are largely taken for granted), and only the Alpha-specific aspect of this work is described here.

This document describes the object programming language pre-processor, the initial distributed program configuration generation program, the file dependency maintenance program, and programs to manipulate executable files (i.e., UNIX *a.out* files). Also described are the initial program loading utilities which are used to move the programs that are compiled on the development workstations onto the testbed nodes and start them running, and the testbed interface programs that run on UNIX and interface to console lines on the processors within the testbed nodes. This report also includes a description of the PROM monitor's command interface, as well as the interface to the testbed's power controller, which is used for turning on and off the individual nodes' power supplies.

2 Alpha Programming Tools

This chapter describes the function of, and user interface to, those programming utilities that were written in support of the general programming needs of the Alpha operating system. The first set of programs described here are ones which are used in the generation of the object files which are to be loaded to and executed on the Alpha testbed. The second set of programs described in this chapter are used to manipulate the standard UNIX representation of executable files.

2.1 Executable File Generation Programs

These programs are used by the kernel-, system-, and application-level programmers to generate executable code for the Alpha operating system. The first utility (occ) is a pre-processor to the C programming language, that provides programming language extensions to the C programming language which provide a syntactically enhanced interface to the services provided by the Alpha kernel. The next programming utility (acp) is used to define the initial configuration of threads and objects for an Alpha application, and generate the load modules for each of the nodes in the specified application program†. Finally, a program (md) was developed to help manage the large collection of (both operating system and application) programs that are associated with Alpha. The standard UNIX *make* program is intended to serve this purpose, but falls short in a number of significant ways when used with large, complex programming systems. For this reason, we developed a higher-level tool to provide input to Alpha's *make*-based programming tools.

The Alpha object language utilities acp (Alpha Configuration Program) and occ (Object language C "Compiler") deal with files that contain Alpha "object language" specification that conform to the defined object language grammar. Such files have the extension ".C". A ".C" file may contain zero or more configuration and/or object definitions. Both acp and occ read a file (i.e., a *map* file) to get the mappings from object language entities (configuration and object names) to their source files.

2.1.1 Object Programming Language Pre-Processor — occ

USAGE:

occ [-h] [-m *mapfile*] [-o *objfile*] [-P] [-q] [-V] [-W] [*switches*] *filename*

-h	—	print usage message
-m *mapfile*	—	specify map file(s)
-o *objfile*	—	object file name(s)
-P	—	do not perform the cc phase of the compilation
-q	—	suppress informative messages
-V	—	print debug output and write ".c" file
-W	—	generate ".c" file
switches	—	switches to be passed on to the cc phase
filename	—	name of ".C" file(s) to be compiled

†*Both acp and occ are described fully in [Shipman 88].*

DESCRIPTION:

The occ program translates ".C" files to ".c" files and runs the C compiler to create a ".o" file. The object language is a simple extension to the C programming language, and is described at length in [Shipman 88]. Warning messages are generated from both occ and *cc*, which refer to source lines in the ".C" file, unless the "-W" switch is used (which causes the ".c" intermediate file to be kept).

occ will attempt to read the object definitions of any objects accessed by the object being compiled, by looking them up in the map file. The more object definitions occ can find, the better a job occ can do of checking the input program. occ will complain if the specifications for referenced objects cannot be found, but it will always produce code (of some sort). Furthermore, occ supports the separate compilation of objects.

It should be noted that the -o switch is different from its normal use in the C compiler. Here it indicates that object names are given as arguments, rather than a file name. Also, occ takes standard C compiler switches and passes them on to the C compilation phase of occ.

FILES:

file.C	—	an object specification file
file.c	—	standard C language file (optionally) generated by occ
file.hdr	—	an object file header (i.e., binary information for use by the loader)
file	—	pure binary executable file, suitable for loading and execution

2.1.2 Configuration Generation Program — acp

USAGE:

acp [-a *applmod*] [-c *confname*] [-h] [-k *kernmod*] [-m *mapfile*] [-n] [-q] [-r *releasedir*] [-R *relpath*]

-a *applmod*	—	the application's *modules* file name
-c *confname*	—	configuration name
-h	—	print usage message
-k *kernmod*	—	the kernel's *modules* file name
-m *mapfile*	—	specify map file to be used
-n	—	do not initialize the build directory
-q	—	suppress the printing of status messages
-r *releasedir*	—	release directory pathname
-R *relpath*	—	relative pathname for the release directory

DESCRIPTION:

The acp program is a utility that allows a programmer to build an Alpha distributed application program to be run on the Alpha testbed hardware. This program takes as input, the Alpha operating system files and the application object definition files and produces a set of Initial Program Load (IPL) images for the Alpha testbed's nodes. These IPL image files constitute the executable initial representation of the distributed program.

In addition to creating the application load modules, acp checks the given configuration for consistency, resolves all references to well-known capabilities by objects, creates data structures that describe object types for the kernel, generates an initialization object for each node, and creates the command files needed to build the kernel, client objects, and boot files for each testbed node.

The distributed program configuration specifications and object type specifications are defined in the Alpha object language definition document [Shipman 88].

The -r switch allows the programmer to specify the path name of the release directory in the command line. The release directory must be specified unless the build directory is a child of the release directory, and the release directory has a name beginning with the characters "*Rel*" (case-sensitive). In that case, acp will infer the release directory's path, unless overridden by the -r switch. If the -r switch is not given and the previous attempt at inferring the release directory's path fails, the program will attempt to infer it based on the name used to invoke acp (in which case the -R switch becomes invalid). The release directory may be specified using either a relative path name or an absolute path name. The difference, however, is significant, in that this path name is passed on to the md program (and the other programs called by acp), and it determines the manner in which all files are referred to in the generated make file. In most cases it is best to use the absolute path name, because the use of relative path names may cause problems when they include symbolic links that span file systems. If a release directory is not given, but is inferred by acp, it will use an absolute path name by default. The -R switch specifies that a relative path name is to be used, however it only applies if the current directory was used to infer the release directory (not the file name of acp itself).

The -m switch is used to specify the path name of the map file, that contains the mappings from an object type or configuration name, to the file name of the file that contains the definition of the object type or configuration. A map file is generated by acp when it initializes the build directory (using results from the md program). Therefore a map file need not be specified when using acp to initialize the build directory.

The name of a configuration can be given to acp with the -c switch, which is then looked up in the map file. If the -c switch is not used, the name of a file containing at least one configuration is expected to appear as an argument. The first configuration defined in that file is built.

The -n switch causes acp to suppress the creation of the initial set of *make* files necessary to build the system. This switch is useful when the initial acp is done manually by the user, but subsequent ones are intended to be done via the *make* program (i.e., using the "*make config*" command).

The Alpha operating system's source file specification (i.e., its *modules* file) is specified with the -k switch. The default file name is "*modules.txt*," and if a file name is given without a directory, acp will first search the release directory's kernel subdirectory, and then the current directory. The application program's *modules* file is given with the -a switch. If this switch is not given in the command line, acp will search the directory where the configuration file is found, for a file named "*modules.txt*," and then it will search for a file with the configuration's name with a "*.txt*" suffix. If both of the these searches fail to uncover a configuration file, the kernel's *modules* file is used. Neither the -k switch nor the -a switch should be used in conjunction with the -n switch.

FILES:

file.C	—	an object/configuration specification file
InitX.C	—	the initialization object for node number *X*, which creates the object instances and initial threads specified in the configuration definition, registers the well-known capabilities defined in the configuration,

and starts the initial threads

file.hdr — load module header for the client object "*file.C*"

modules.txt — file containing the names and locations of a collection of source files

kernappl.map — file of mappings between object names and object specification files

fixlmhdr — utility program that inserts into the load modules' headers the correct sizes of all of the configuration's objects, once they have been compiled, linked, and padded out to page boundaries

KOTtable.c — kernel object table definition file—a C source file that contains the initialized kernel object table (which contains information about the objects and well-known capabilities in the configuration)

oprindices.c — operation indices definition file—a C source file that contains the initialized declarations of the global variables that provide the mapping from object-name/operation-name or well-known-capability-name/operation-name pairs to operation indices used by occ for invocations

make.make — configuration switches and definitions (included by the master make file, i.e., *makefile*)

conf.make — rules for creating dependency relationships file for the configuration

makefile — master make file for the configuration

2.1.3 File Dependency Maintenance Program — md

USAGE:

md [-c] [-d *defdir*] [-f] [-m *module*] [-p] [-r] [-s] [-v] [-w] [-4]

-c — complain if the declared include files do not match the actual ones

-d *defdir* — override the default directory

-f — use built-in make rules rather than running the make file through *m4* (this is the default)

-m *module* — output file name

-p — check the declared include files on a per-module basis

-r — do not generate rules and dependencies, only the file name macros

-s — generate a separate makefile for each module

-v — enable the printing of status information

-w — write out the *m4* input file

-4 — run the make file through *m4*, rather than using the built-in rules

DESCRIPTION:

This program takes a list (in human readable form) of the source files that comprise a program, and creates the inter-file dependency specifications for use by the UNIX *make* utility. The md program looks at a specified group of files, derives all the dependencies implied by inclusion (via the *cpp* "*#include*" commands) of other files, and generates *make* commands that specify those dependencies, along with macros defined conveniently for various meaningful sets and subsets of the files specified, and rules for compiling the source files.

A particular run of md works on a collection of files specified in a modules specification file ("*modules.txt*" is the default name). This collection is organized in terms of "*modules*," which is defined as a collection of related source files. A source file is defined to be any file that is transformed by applying a make rule into a UNIX ".o" file.

The types of source files (as indicated by their characteristic suffixes) that are currently supported by md include:

- .c — C code
- .a — preprocessed assembly language
- .s — standard assembly language
- .l — UNIX *lex* specification
- .y — UNIX *yacc* grammar specification
- .C — Alpha object language code (a special case)

FILES:

conf.make — a file including all of the file dependencies for the *conf* configuration (to be included in a master *make* file)

2.2 Executable File Manipulation Programs

These programs allow files that are in the standard UNIX object format (i.e., *a.out* files) to be used for purposes other than execution under UNIX. These tools are needed in order to make the UNIX workstation an acceptable environment for cross development support. The first of these programs (seg68), is used to remove the individual code and data segments from the *a.out* file, in order to generate a "pure" executable object file. The next program (down68) is used to convert *a.out* files into Motorola S-record format, for use with serial down-line-load programs and communicating with PROM programmers. The last of these programs (prom68) splits an object file into odd- and even-byte streams, for the programming of byte-wide PROMs.

2.2.1 Extracting Code and Data Segments — seg68

USAGE:

seg68 [-o *outfile*] [-P] [-t [b [d]]] *infile*

-o *outfile*	—	output file name
-P	—	pad all segments to page boundaries
-t	—	emit the input file's *text* segment
-b	—	emit the input file's *bss* segment
-d	—	emit the input file's *data* segment
infile	—	the *a.out* format file to be used as input

DESCRIPTION:

This program is used to strip the header, relocation, and symbol table information from UNIX *a.out* files, in order to create pure, executable files, consisting of one or more of the input file's (potentially padded) segments.

If the -o switch is not used to indicate a file to place this program's output into, the output is generated on standard output. The -P switch is used to pad out all emitted segments (with 0 values) to ensure that all of the following segments are page-aligned. Any combination of the -t, -d, and -b switches can be used together to generate the desired output file (at least one of these switches must always be specified). The segments are always emitted in the following order: *text*, *data*, and *bss*. If the -P switch is used with the -db switch combination, the *data* and *bss* segments are concatenated and padded as a single segment. All other alignment of segments is derived from the input file (i.e., depending on the *a.out* file's magic number).

This program works with both the old (i.e., Sun-2) and the new (i.e., Sun-3) *a.out* file formats, but only generates code for Sun-2 processors (i.e., 2KB pages).

FILES:

a.out — a standard UNIX executable file (see the UNIX man page for *a.out(5)*)

2.2.2 Creating Motorola S-Records — down68

USAGE:

down68 [-e *address*] [-m] [-o *file*] [-T *address*] *filename*

-e *address*	—	base address for start of execution
-m	—	generates S1 records
-o *file*	—	name of output file
-T *address*	—	base loading address
filename	—	name of input file (in *a.out* format)

DESCRIPTION:

This program is used for converting UNIX *a.out* files into Motorola S-record format. S-records format is an ASCII hexadecimal representation of arbitrary binary files. All records have a similar format—the record type (S*x*, where *x* is a digit from 0 to 9), the data (character pairs of hexadecimal digits), and a checksum (four hexadecimal digits). There are multiple types of S-records, including:

- header records (S0) — that contain the name of the input file (in ASCII)
- text or data segments records (S1 or S2) — that contain the input file data
- start of execution records (S8) — that contain the starting address for execution
- trailer records (S9) — that indicate the final record in the file

The default is to generate S2 records, but if the -m switch is used, this program generates S1 format records instead. The -T and -e switches are used to override the values found in the input file's *a.out* header.

S-record format files are used for both the serial-line down-loading to the testbed processors and for programming PROMs.

FILES:

a.out — a standard UNIX executable file (see the UNIX man page for *a.out(5)*)

2.2.3 Splitting S-Records Into Odd and Even Bytes — prom68

USAGE:

prom68 [-h | -l] *filename*

-h	—	emits high (i.e., even) bytes
-l	—	emits low (i.e., odd) bytes
filename	—	name of input file (in S-record format)

DESCRIPTION:

This program is used to split files (that are in S-record format) into odd and even bytes for use with a PROM programmer. The given input file is broken into either high or low bytes, depending on whether the -h or -l switch is used, and the output is sent to standard output.

If no switch is given, the program defaults to a mode that generates high bytes.

FILES:

a.out — a standard UNIX executable file (see the UNIX man page for *a.out(5)*)

3 Testbed Processor Initial Program Loading Tools

The programs described in this chapter are used for loading the testbed processors with object code generated on the development workstations. The first of these loading programs (sdl) takes a file in a record-oriented ASCII representation, and ensures its reliable transfer across a serial-line from the testbed's gateway machine to a testbed processor. The next program (tftpd) allows the testbed's Ethernet to be used to both up- and down-load arbitrary data between the gateway machine and a testbed processor's memory. The last of this set of programs is used in the process of loading initial programs into the testbed nodes (IPLloader) and in the creation of boot image files to be used for this purpose (mkboot).

3.1 Down-Loading via the Console Serial Lines — sdl

USAGE:

sdl *filename*

filename — file (in S-record format) to be down-loaded

DEFINITION:

This program is used in conjunction with the PROM monitor's l command (as described in Chapter 5) to load a testbed processor's memory via the serial-line. This program runs under UNIX, opens the file specified in the command line, and sends the contents of the file, one record at a time, while handshaking with the target processor (using both positive and negative, per-record acknowledgments).

Once the l command is issued on the target processor, the monitor waits for a line of ASCII information in Motorola S-record format. When one is received, the PROM program validates the checksum for each record, loads the body information into the processor's memory at the indicated location (if the record is valid), sends an acknowledgment, and repeats this procedure. This activity continues until a "go" record is received, at which point the target processor begins execution at a given location.

The down68 program (described in Chapter 2) is used to convert a UNIX *a.out* file into Motorola S-record format.

The sdl program takes the data input from a specified file, and uses the UNIX *stdin* and *stdout* streams for interaction with the target processor. Typically, this program is executed from within the UNIX *tip* program. The *tip* program's "~C" command is used to start up the sdl program. This connects the *tip* program's serial-line connection to a target testbed machine with the *stdin* and *stdout* streams of sdl.

FILES:

/etc/remote — contains the specifications for the serial lines which connect the testbed processors' console lines to the gateway machine (see the UNIX manual page for *remote(5)*), used by *tip*

/usr/bin/tip — a UNIX program that provides a connection to an I/O device and permits the user to carry on an interactive terminal session with the connected device

3.2 Up-/Down-Loading via the Ethernet — tftpd

USAGE:

tftp [-d]

-d — enables debugging output

DESCRIPTION:

The DARPA standard Trivial File Transfer Protocol (TFTP) is used to move bytes between the testbed processor's memory, and files on any machine that implements the protocol. The tftpd program is the TFTP server daemon that runs under UNIX, in support of the protocol. The other half of this program is the user-level client program that runs on the testbed processors. The tftpd program responds to file transfer requests from the testbed processors, initiated by the PROM monitor's **J** command (as described in Chapter 5). The TFTP client and server programs are used together to both load a testbed processor's memory from a development workstation's file system, and to load a file on a workstation from the processor's memory.

In the Alpha testbed, the gateway machine serves as a TFTP server and responds to all TFTP requests from the processors in the testbed. For this to work, the gateway machine must be properly configured—i.e., the network device for the testbed network must be up and configured (via the *ifconfig* utility), the Internet Protocol (IP) handler must be set up to allow the tftpd daemon to be invoked when TFTP packets arrive (via the *inetd* daemon and the */etc/servers* and */etc/services* files), the gateway machine's */etc/network* and */etc/hosts* tables must be set up with the proper host names and IP addresses, the gateway's ARP daemon must be able to respond to address resolution requests from the testbed processors, and the proper access permissions must be set on the gateway to allow the desired files to be read and written (because the tftpd program operates with *uid* and *gid* of -2).

The standard SunOS release of the tftpd has been defective, and thus a working version had to be written and substituted for the distributed one. It is possible that the defect in the standard program may be corrected in subsequent SunOS releases, which would eliminate the need for this customized program. For a complete description of the Sun version of the tftpd program, see the UNIX manual page for *tftpd(8C)*.

FILES:

/etc/hosts — contains host name to IP address mappings (see the UNIX manual page for *hosts(5)*), and should include the IP addresses of all of the processors in the testbed

/etc/networks — contains mappings of network name to network field of an IP address (see the UNIX manual page for *networks(5)*), and should include the IP address (used by the gateway machine) of the testbed's network

/etc/services — contains port number and protocol service type for tftpd (see the UNIX manual page for *services(5)*)
the default port number for TFTP is 69, and TFTP uses UDP datagrams

/etc/servers — contains the pathname for the tftpd executable image to be invoked when a TFTP packet arrives at the port given in */etc/services* (see the UNIX manual page for *servers(5)*)

3.3 Boot Loading the Processors within a Node — mkboot and IPLloader

USAGE:

mkboot [-o *ofile*] [-l *lfile*] [[-p *procnum*] [-s *startaddr*] *filename*] …

- -o *ofile* — put output into *ofile*
- -l *lfile* — use the loader program in *lfile*
- -p *procnum* — target processor number for the associated load file
- -s *startaddr* — starting address for the associated load file
- *filename* — name of an individual load file to use as input

DESCRIPTION:

This program takes as input a collection of executable files (in UNIX *a.out* format) and generates a file that can be used to load the multiple processors in an individual Alpha testbed node. The PROM monitor code in the testbed processors includes an initial program loading (IPL) mechanism that makes use of files generated by the mkboot program.

This program is intended to be a general-purpose boot-load facility, and to this end, the output of the mkboot program can be recursively used as its input to construct hierarchical boot load files. That is to say, an input file to this program can be either a simple executable file (in *a.out* format) or a load module generated by mkboot itself.

The output file generated by each run of mkboot includes a loader program that interacts with the PROM code in the target processor(s) in the course of completing the IPL. In the loading of the constituent processors in a node, the loader program is responsible for the relocation of an individual load module's code and data and for managing interactions with other processors being loaded. If a loader program is not specified with the -l command, a default program, named IPLloader (whose path is compiled into the mkboot program) is used.

In the current Alpha testbed, the node IPL procedure begins with the down-loading (with either TFTP or sdl) of an IPL file onto one of the processors within the node to be loaded. Once loaded, execution is begun at the start address of the loaded module (where the (top-level) IPLloader program exists). The IPLloader program uses the information in the configuration page of the boot image file to determine the sizes and locations of the various parts of a load module. Based on this information, the IPLloader program then distributes the top-level load module parts to the specified processors. When this distribution is complete, the IPLloader either begins execution at the specified load point in the (potentially relocated) local load module, or, if there is no local load module, returns to the command level in the PROM monitor.

A processor number, starting address, and load file name is specified for each processor to be loaded by the output load module. If the -p switch is not given, a default order (that starts with 0 and is incremented by one for each new load file) is used. Also, if the -s switch is not given, the associated load file is loaded at a default location (as specified within the mkboot program — currently 0x4000). This information is placed into the output load module's configuration table, and can be ignored by other programs.

Figure 1 illustrates the high-level structure of the IPL files generated by mkboot.

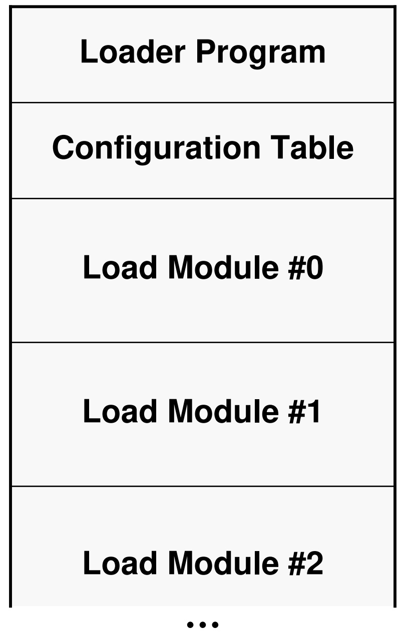

Figure 1: Initial Program Load File Structure

All of the constituent components of an IPL file are guaranteed to be padded out to sizes that are multiples of Sun-2 pages (i.e., 2KB). This can be done by processing the load files with the `seg68` program.

FILES:

IPLloader.c— the default IPL loader program that works with the IPL code currently within the testbed processors' PROM monitor code

4 Testbed Interface Tools

These programs provide the programmer with the basic interface to the console serial-lines of the processors in the testbed nodes. The first of these tools described here is a window-based utility (nodetool) that provides a user with a general-purpose, programmable interface to the collection of processors that comprise an Alpha testbed node. The other utility described here (rcons) provides a more basic interface to the individual testbed processors, in the form of a remote console facility.

Both of these tools provide a mechanism for controlling access to the testbed, and they allow interactive execution, control, and monitoring sessions to be performed on the testbed processors via workstations anywhere on the Internet. In addition, both of these tools allow the scripting of interactive sessions (i.e., the transcription to a file of the character streams generated by the user and the target machine).

4.1 Node Access via SunView — nodetool

USAGE:

nodetool [-i *initfile*] [-p *X Y*] [-h *height*] [-n] [-v] *nodename*

-i *initfile*	—	use *initfile* as the program's initialization file
-p *X Y*	—	initial position of the upper left corner of the window
-h *height*	—	set the console panel height
-n	—	ask for confirmation on exit
-v	—	print status messages
nodename	—	name of the node

DESCRIPTION:

The nodetool program is a SunView-based program that provides a user with an interactive interface to the processors within the Alpha testbed. This program creates a window, with a control panel and a separate console panel for each processor in the node configuration specified in the start-up file. Figure 2 provides an example of a nodetool window with a control panel and three console panels, all in default configurations.

The defaults for the initial size and position of the window can be overridden by way of the program's command line switches. Also, the initial location the window and the sizes of the panels can be altered while the program is running.

The configuration of the control panel and the testbed processors to which the console panels are connected is user-programmable and is defined by a configuration file that is read at program start-up time (and can be read in again while the program is running). The default configuration file is called "*.rcnode*" and is to be located within the directory from which the nodetool program was invoked.

<u>Window Layout</u>

The window created by the nodetool program has three major components—a title bar, a control panel, and a set of console panels. The nodetool window's title bar contains the logical name of the node configuration being run (as given in the command line's "node-name" parameter), and an indication of the current status of the tool (e.g., ready, printing, waiting for a match, reloading, quitting).

The control panel contains a pair of string variables and collection of buttons, all of which are user-configurable. The user can configure the control panel by specifying the contents of variables, button labels, and the actions that are performed when a button is "pushed." There are two standard buttons that always appear on the control panel, and are not programmable, they are: the QUIT button, that resets the windows and then exits; and the RELOAD button, that causes the program to re-read the program's configuration file and resets both the control panel and the console panels.

The control panel has eight programmable buttons in the standard configuration, each of which can be given arbitrary labels and assigned arbitrary actions to perform when the button is activated. The actions that can be performed as a result of a button push emulate a user's interactions by way of a series of commands that alternately emit character strings to a connected processor and await specific character patterns from a target processor. In addition to being able to specify arbitrary character patterns to be emitted, or matched, buttons can be configured to delay a given number of milliseconds between the individual characters it emits, or to wait a given number of milliseconds for a given character string match to occur. Should a match action time-out (i.e., the desired character pattern is not received in the given amount of time), the affected console panel blinks and the program continues with the execution of the button's action specifications. Furthermore, each of a button's individual actions may be directed at any one of the processors attached to a given console panel.

There are also a pair of user-definable variables in the control panel, that provide two different presentations of their contents. The first of these variables is an editable string known as the "value", and the other variable is known as the "option", and consists of a collection of strings that can be cycled through by clicking its label with the mouse. Both the value and option variables have user-definable labels associated with them on the control panel. The control panel variables can be used in both printing and matching types of button actions.

Typically, one console panel is created for each processor within a node, and the chosen testbed processor is connected to a console panel in such a fashion as to emulate a true hardwired connection to the given processors' serial ports. The console panels have scroll bars on the left edge that can be used to move around in the history log kept for each console panel. The break command is issued to a processor by typing a '~' character to the console panel (this is the same command as used in the rcons program).

Program Operation

The configuration file consists of a collection of logical node specifications, that define the program's configuration for a particular set of testbed processors. Each individual node configuration in a file has a logical identifier associated with it that is used to select the configuration on program start-up. A node configuration contains a list of processor specifications that indicate the testbed processors that are to be associated with console windows, an (optional) initial string to be issued to the specified processor on connection, and an (optional) script file specifications.

If a script file specification is given in a processor specification, there is a choice of appending or creating a new script file (by use of the **a** and **w** script mode switches, respectively), and the complete interactive session is recorded in the given file. The scripting remains in effect until the program is exited or scripting is explicitly disabled by a button action. Among the actions which can be performed as a result of a button push are the enabling and disabling of scripting for given console panels. Scripting can be turned on and off for a given window any number of time in the course of an interactive session.

The order in which the processor specifications appear in the node configuration defines the processor number used in the button action specifications to refer to specific console windows.

The nodetool program creates a console panel for each node specification that it finds in the specified node configuration, it then attempts to connect to each of the specified testbed processors (via the rconsd daemon). The program generates an error and exits if connections cannot be made to the specified processors. Once connected to a process, the program issues the start-up string, if one has been specified. Following this, the program attempts to initialize the control panel. It reads the variable and button specifications and initializes them. Should syntax errors be detected in the reading of the configuration file, the program will print an error message and exit. Following the successful completion of these initialization steps, the program is ready to accept user input. Throughout this sequence of operations, the status of the procedure is displayed in the window's title bar.

There are certain (compile-time-defined) restrictions on the nodetool. For example, a maximum of three console panels can be created, print strings and match strings are limited to 80 characters, and a maximum of 25 individual commands can make up a button's action specification. Each of these limits can be altered by changing constants and recompiling the program.

Configuration File Format

The following is a description of the syntax for the node configuration file. In this description, the keywords appear in boldface, while the atoms appear in italics. Optional terms appear in "{ }?", one or more occurrences of a term appear in "{ }+", zero or more occurrences of a term appears in "{ }*", and alternate choices are separated by the '|' character.

configuration_file	::=	{ node_spec }+
node_spec	::=	node_header { processor_spec }+ variable_spec { button_spec }+
node_header	::=	**NODE** node_name
node_name	::=	*string*
processor_spec	::=	**PROCESSOR** processor_name { startup_string }? { **SCRIPT** script_mode script_filename }?
processor_name	::=	*string*
startup_string	::=	*quoted_string*
script_mode	::=	{ **a** | **w** }
script_filename	::=	*string*
variable_spec	::=	{ value_spec }? { option_spec }?
value_spec	::=	**VALUE** value_label value_string
value_label	::=	*quoted_string*
value_string	::=	*quoted_string*
option_spec	::=	option_header { choice_spec }+
option_header	::=	**OPTION** option_label

option_label	::=	*quoted_string*
choice_spec	::=	**CHOICE** choice_string
choice_string	::=	*quoted_string*
button_spec	::=	button_header {action_spec }*
button_header	::=	**BUTTON** button_number button_label
button_number	::=	*integer*
button_label	::=	*quoted_string*
action_spec	::=	{ print_string_spec | print_variable_spec | match_string_spec | match_variable_spec | script_on_spec | script_off_spec }
print_string_spec	::=	**PRINT_STR** processor_number delay_value print_string
processor_number	::=	*integer*
delay_value	::=	*integer*
print_string	::=	*quoted_string*
print_variable_spec	::=	**PRINT_VAR** processor_number variable_number
processor_number	::=	*integer*
variable_number	::=	*integer*
match_string_spec	::=	**MATCH_STR** processor_number delay_value match_string
processor_number	::=	*integer*
delay_value	::=	*integer*
match_string	::=	*quoted_string*
match_variable_spec	::=	**MATCH_VAR** processor_number variable_number
processor_number	::=	*integer*
variable_number	::=	*integer*
script_on_spec	::=	**SCRIPT_ON** processor_number script_mode script_filename
script_off_spec	::=	**SCRIPT_OFF** processor_number

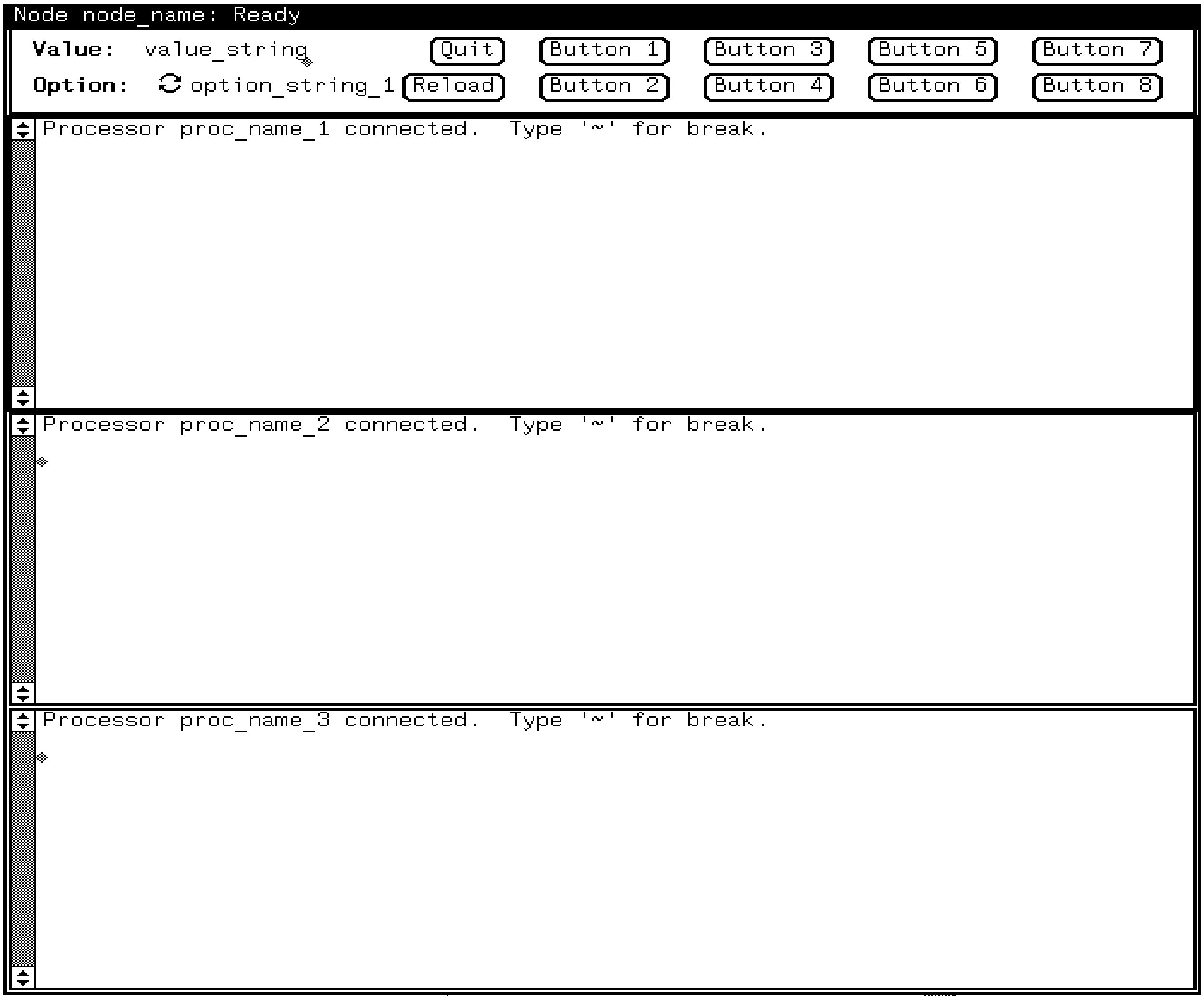

Figure 2: Example nodetool Window

FILES:

./.rcnode — the node configuration file, containing the specifications for the processors that the tool is to connect with and both the labels and actions associated with the tool's programmable buttons

4.2 Node Access via Workstations

Each of the Alpha testbed's processors are connected to the gateway machine via serial lines. These console lines are the primary means by which the users of the testbed interact with the processors which make up the testbed. By way of these serial-line connections, a user can reset the processors, and load, control, and monitor the code running within them.

The UNIX *tip* program (as described in the UNIX manual page for *tip(1C)*) provides the basic functionality needed to allow a user to carry on an interactive session with one of the processors. However, the *tip* program has a number of limitations, the foremost of which is the fact that it must be executed on the machine which has the physical serial-line connections (i.e., the gateway machine in the Alpha testbed). In order to permit access to the testbed processors from any machine on the Internet, a "remote console" facility was created. This program consists of two components—a client program (rcons), that can be executed on any development workstation, and a server program (rconsd), that must be running on the gateway node. The following subsections describe these two programs and their user interfaces.

4.2.1 Client Program — rcons

USAGE:

rcons [-k] [-l *logfile*] *name*

-k	—	kill the named rconsd and exit the program
-l *logfile*	—	append all console output to the file named *logfile*
name	—	logical identifier of the console to be connected to (must be one of the names in the *rconsoles* file)

DESCRIPTION:

The rcons program is used by a client to access the hardwired serial-lines that connect the gateway machine with the testbed consoles. The intent of this program is to provide the functionality of the UNIX *tip* program for remote users.

The -l switch allows users to create transcripts of their interactive sessions, while the -k switch allows users to terminate a given server process.

The *name* parameter given in the command line must correspond to one of the server names (or aliases) found in the *rconsoles* file (which is described in the following subsection). Furthermore, for a connection to be successfully made, a server corresponding to the logical name given in the command line must be active on the gateway machine, and must not already be in use. If the logical name is invalid, the server is not accessible, or the desired server is in use, the rcons program will exit an error message will be given, indicating the cause of the failure.

Once successfully connected to an rconsd server, the rcons program accepts the following commands (where '^' is a prefix for a control character):

^_	—	disconnect from the rconsd and terminate the program
~	—	send a break to the console line

With the rcons program it is possible for a user to develop system software, remote from the physical location of the testbed machines. In fact, because rcons uses DARPA Internet standard protocols, development work can be performed equally well from anywhere on the Internet.

4.2.2 Server Program — rconsd

USAGE:

rconsd [-a] [-m] *name* &

-a	—	causes the daemon to require a password from the user for access to the console line
-m	—	causes the daemon to keep the console serial-line locked
name	—	logical identifier of the console to start a daemon for

(must be one of the names in the *rconsoles* file)

DESCRIPTION:

The rconsd program is a process that runs on the node with the physical device and provides an interface between individual activations of rcons programs and one of the testbed's serial-lines. This program obeys the UNIX *uucp* locking protocol and is compatible with *tip*, *uucp*, and other *rconsd* programs.

The -a switch requires that a password be given for access to the serial-line that an rconsd program is serving. This makes it is possible to restrict access to the testbed machines from the entire DARPA Internet community (which is the default). This feature is particularly useful for such devices as the power controller, unrestricted access to which could allow physical damage to be done to the testbed hardware.

The -m switch causes the serial-line to be locked and remain locked as long as the rconsd daemon is active. This option is used to restrict access to the testbeds via the *tip* program.

The *name* parameter given in the command line when starting up an rconsd program specifies the serial-line device for which the program is to provide access service. This logical name is the same as that given to the rcons program by the user, and must also appear in the console description file.

The console description file contains a description of each device for which remote console service can be provided. Each potential console is described by a single line in the console description file in the following format:

<***tty***> <***baud***> <***server_host***> <***port***> <***common_name***> [***alias_list***]

tty	—	UNIX pathname of the device file (i.e., */dev/ttyNN*)
baud	—	baud rate code for the tty line (see the UNIX file *<sgtty.h>* or the UNIX manual page for *tty(4)*)
server_host	—	name of the host on which the server runs
port	—	Internet port number on this host associated with this particular daemon (this may be privileged (i.e., < 1024) to keep unprivileged daemons from being run)
common_name	—	name most commonly used to refer to a console
alias_list	—	list of other names that can be used to refer to a console

The console description file is named *rconsoles* (by default) and located in a directory that is defined in the remote console facility's *makefile*. Because the rconsd server uses the standard UNIX IPC facility, it is possible for client programs other than rcons to make use of the remote console service (e.g., the node tool, described in the previous section, uses the rconsd servers to manage its remote connections to the testbed processors).

FILES:

/dev/ttyXX	—	serial line devices (make sure protection is right)
/etc/ttys	—	default definitions for all tty lines on a node (make sure that logins are disabled for the console lines)
/etc/rc.local	—	includes commands for start-up of each rconsd daemon
rconsoles	—	list of consoles
/etc/services	—	contains port number for rconsd IPC interface

5 Testbed Processor PROM Monitor

The Sun processor boards used in the Alpha testbed contain on-board PROMs, within which resides monitor code. In the Alpha testbed, the monitor provides: the basic hardware initialization on power-up, simple I/O functions, and software debugging and execution control support. The monitor commands can be executed interactively via the processor's console serial-line, or at run-time, by calling the subroutines (from code which must be executing in supervisor mode). It should be noted that the following discussion applies to the Sun version 2.0 processor board. The differences between the monitor command interface described here and that provided by the version 1.5 board are minimal, and are fully documented in [Sun 84].

5.1 Peek and Poke Memory Commands — B, L and W

These commands are used to examine and modify memory locations, in units of one, two or four bytes (i.e., byte, word, or long accesses).

Open Byte Memory Location	—	**B** [*address*] [*actions*]
Open Long Memory Location	—	**L** [*address*] [*actions*]
Open Word Memory Location	—	**W** [*address*] [*actions*]

The memory peek/poke commands take the hexadecimal address of a memory location (i.e., *address*), prints its contents (also in hexadecimal format), and then processes further input. At this point, the contents of the opened location can be modified by typing a hexadecimal value, or the location can be closed and the following location opened by typing a carriage return. It is also possible to return to the monitor's command level by typing a non-hexadecimal character at this point. If no address is given with one of these commands, location 0 is opened. If an odd address is given for a long or word access, the long/word at the lower address is opened. Any commands provided in the optional *actions* field are executed following the opening of the specified memory location.

5.2 Register Access Commands — A, D and R

These are commands that can be used to read and write the processor's data, address, and system registers. There are eight data registers and eight address registers in the MC68010.

Open Address Registers	—	**A** [*number*] [*actions*]
Open Data Registers	—	**D** [*number*] [*actions*]
Open System Registers	—	**R** [*actions*]

These commands open CPU registers—i.e., they print the contents of a register, and permit the contents to be altered. In the case of address and data registers, the commands take an optional starting register number (zero through seven, the default being zero). As with open memory locations, the registers can be modified by typing a hexadecimal value, or the open register can be closed and the following register can be opened by typing a carriage return. Control can be returned to the monitor's command level by typing a non-hexadecimal character.

The system registers include the supervisor-mode stack pointer (SS), the user-mode stack pointer (US), the source address space field (SF), the destination address space field (DF), the exception vector base (VB), the supervisor-mode context (SC), the user-mode context (UC), the system status register (SR), the program counter (PC).

5.3 Memory Management Unit Manipulation Commands — M and P

These commands allow the tables that make up the processor's memory management unit to be manipulated.

Open Segment Map	—	**M** [*address*] [*actions*]
Open Page Map	—	**P** [*address*] [*actions*]

These commands take a virtual address and open for manipulation the segment or page map for that address. The **M** command opens the segment descriptor for the given address, within the current context's segment map (as defined by the current value of the context register). The value of the segment descriptor is displayed and it can be altered by typing hexadecimal characters. The open descriptor can be closed and the descriptor of the next segment can be opened by typing a carriage return, and control can be returned to the monitor's command level by typing a non-hexadecimal character.

The **P** command behaves much the same as the **M** command, except it opens a page descriptor for the given address, using the value of both the current context register and the current segment descriptor for the given address. With either command, if no address is given a default value of 0 is used.

5.4 Execution Control Commands — C and G

These commands are used to instruct the processor to begin execution at some location.

Continue Execution	—	**C**
Begin Execution at a Location	—	**G** [*address*]

The **C** command is used to continue execution at the point defined by the PC and in the state indicated by the SR (as indicated by the register access commands). The **C** command is typically used to resume execution following the execution of a trap or the issue of a break.

The **G** command begins processor execution at the specified hexadecimal address, using the MMU and register values indicated by the MMU and register access commands. If no address is given, execution is started at the address given in the last **G** command (or location 0x2000 if no **G** command was issued since processor initialization).

Break	—	~
Reset Processor State	—	**K** [*type*]
Alter CPU Address Space	—	**S** [n*umber*]
DUSART Manipulation	—	**U** [*option*]

5.5 Debug Commands — T and Z

These are commands that can be used to perform low-level debugging of code.

Modify Instruction Tracing	—	**T** [*switch*] [*cmdlist*]
Manipulate Instruction Breakpoint	—	**Z** [*address*]

The **T** command is used to enable the MC68010's trace trap bit in the SR, which causes the processor to execute a trace trap following the completion of each instruction. This is useful for single-instruction-cycle debugging. The **T** command takes both an optional switch and a command list. The legal switches are: **y** — to turn tracing on; **n** — to turn traceing off; and **c** — which turns on tracing and executes commands from the given command list at the end of each instruction trap. Following each trap, the address and opcode of the executed instruction is displayed.

The **Z** command is used to put a breakpoint trap at a given address. This is done by replacing the contents of the specified memory location with a trap #1 instruction and saving the previous value of the location. When the processor executes the trap instruction, control is returned to the monitor's command level and the monitor commands can be used to determine the state of the processor's registers, memory, and MMU. If no address is given with the **Z** command, the monitor displays the address of the currently defined breakpoint (if a breakpoint has not been established since processor initialization, the address 0 is displayed). If the address 0 is given with the **Z** command, the currently installed breakpoint is removed and the previous value of the location is restored. When a valid (non-zero) address is given with the **Z** command, the monitor installs the breakpoint trap instruction and prints the value of the given location that was overwritten. The monitor only maintains a single storage location for the instruction displaced by the breakpoint trap, so if a breakpoint is installed when another **Z** command is issued, the previous breakpoint is removed and the new one is added. Note that if the processor is soft-reset, the address and value of the displaced instruction is lost and the trap instruction remains in place.

5.6 Miscellaneous Commands — K, S and U

These commands are used to reset the processor, alter the processor's address space, and manipulate the on-board DUSART.

The break command (~) is used to interrupt the execution of the processor and enter the monitor's command level. The break command is issued with an out-of-band signal on the console serial-line, generated by generating a mark signal of a duration longer than a standard character frame time. The on-board DUSART detects break signals and the monitor code polls the DUSART's control register and executes a trap command when a break is detected.

The **K** command is used to soft-reset the processor. The *type* parameter defines the kind of reset to be done: **0** causes the SSP to be set to 0x1000; **1** resets the SSP and initializes the CPU and the other major components on the processor board; **2** (or greater) resets the whole processor board, clears primary memory, and executes diagnostics; and **b** performs no reset but prints the processor's start-up banner. If a non-hexadecimal *type* argument is given with the **K** command, a value of 0 is used as a default. The **K 2** command performs a hard-reset similar to that which is done by the processor on power-up. On reset, the PROM-based diagnostics test primary memory, the CPU, the MMU, the timers, and the DUSART.

The **S** command is used to set the current address space to the value given in the *number* argument. If an invalid address space number is given (i.e., anything but a hexadecimal number from 0 through 7), the current address space is printed by the monitor.

The **U** command is used to manipulate the on-board DUSART. The serial-line console uses the A port, and the B port is not used by the monitor. Most of the options for this command apply to either the A or the B port. If no option is given, the current state of the DUSART's settings are printed.

The **U** command's options are:

{a | b}{i | o}	— Choose port A/B as input/output device
k{i}	— Use keyboard for input
s{o}	— Use screen for output
ks | sk	— Use keyboard for input and screen for output
{a | b}#	— Set port A/B speed
{a | b}t	— enter transparent mode with port A/B
t	— enter transparent mode with port B
esc **.**	— exit transparent mode
esc **#**	— send break to connected port
e	— echo input to output
ne	— disable echo input to output
x *char*	— set escape character to ***char*** (defaults to ^^)
f	— enable XON/XOFF (^S/^Q) flow control
nf	— disable automatic flow control

5.7 Up-/Down-Load Commands — I and J

These commands are used to load code or data into the processor's local memory from an external load source.

Start Serial-Line Bootload	—	**I** [*port*]
Start Ethernet TFTP Program	—	**J**

The **I** command is used to load the processor's memory via the serial-line. The *port* parameter is used to indicate which port (A/B) the information is to be loaded on. Once the **I** command is issued, the monitor waits for ASCII information in Motorola S-record format, validates the checksum on each record, loads the memory at the indicated location (if the record is valid), sends an acknowledgement, and continues waiting until it gets a "go" record, at which point it starts execution at the desired location.

The **J** command is used to both load the processor's memory from a workstation's file system, and to load a file on a workstation from the processor's memory. The **J** command uses the DARPA standard TFTP protocol [Sollins 81] to move bytes between the processor's memory and files on any machine that implements the protocol.

Command options and sub-commands for the **J** command are:

- **h** — modify host table
  - ' ' — print current host address
  - *q* — return to the command level
  - *?* — print help message
- **r** — download a file
  - *start* — starting memory address
  - *size* — number of bytes
  - *filename* — UNIX pathname of file
- **s** — upload a file
  - *start* — starting memory address
  - *size* — number of bytes
  - *filename* — UNIX pathname of file
- **q** — quit
- **v** — enable verbose mode
- **?** — print help message

6 Testbed Power Control Tool

The AC power provided to each node in the testbed can be (individually) switched on and off by way of a custom-built unit known as the testbed's *power controller*. The power controller is based on a single-chip microprocessor (with a ROM-based BASIC interpreter) that provides a user-interface and controls the power switching electronics.

The power controller's current user-interface program provides the following set of commands:

Print Help Message	—	**H**
Print Status of All Nodes	—	**S**
Turn All Nodes On	—	**I**
Turn All Nodes Off	—	**O**
Turn an Individual Node On	—	**i [*node #*]**
Turn an Individual Node Off	—	**o [*node #*]**
Reset an Individual Node	—	**r [*node #*]**

The power controller is capable of controlling up to eight testbed nodes. The power controller is connected to the testbed's gateway processor via a serial line, and is accessible from the Internet by way of either the tip or rcons programs.

While it was under development, the power controller's program was not committed to PROM. Instead, a simple download program (pwrdl) was used to download the machine via its serial line interface. This can be accomplished by typing the following UNIX command:

```
pwrdl < pwrclt.b > /dev/ttyXX
```

This command must be given on the gateway node and the appropriate string must be substituted for the "XX" (to indicate the serial-line to which the power controller is connected).

FILES:

pwrctl.b	—	the BASIC program executed by the power controller
pwrdl.c	—	the source code for the power controller down-loading program
pwrdl	—	the executable power controller down-loading program

7 Acknowledgments

Most members of the Alpha team contributed in one way or the other to the project's support tools. In particular, Duane Northcutt created the initial versions of the object language preprocessor, while Sam Shipman modified, then rewrote them, creating the current versions of these programs. Similarly, Duane created the original programming environment support tools (using the *make*, *awk*, *sed*, *grep*, *ed*, *m4*, etc. utilities), then Sam modified and enhanced these tools and ultimately created the MD program that replaced much of the functionality of the earlier tools. In addition, Sam created the ACP program to automate the (previously manually performed) generation of distributed program configurations.

Duane created the seg68 program, modified the standard UNIX loader (*ld)*, and modified the Sun processor's PROMs. The down68 and prom68 programs were modified by Duane, based on similar programs written by Raphael Bracho of the CMU Robotics Institute. Furthermore, Duane created and programmed the testbed power controller.

Ray Clark and Duane created the device driver used by the gateway machine to connect serial lines to the testbed nodes, the UNIX *tftpd* program, the PROM-resident TFTP client code, along with the UNIX and PROM code for the serial download program (sdl).

Huay-Yong Wang, and Ray implemented the mkboot program.

Sam created the rcons and rconsd programs, starting from code originally written by James Wendorf.

Duane wrote the window-based testbed node interaction tool (node), and the interface was based on a similar (special-purpose) tool built by Pete Keleher at General Dynamics.

Duane designed and implemented the initial version of the power controller hardware and its interface software, and Bruce Taylor re-implemented the unit.

Both David Maynard and Jeff Trull contributed to this effort by creating, installing, and maintaining various other support utilities.

UNIX is a trademark of AT&T Bell Laboratories.

Ethernet is a trademark of Xerox Corporation.

SunOS is a trademark of Sun Microsystems.

References

[Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April, 1988.

[Sollins 81] Sollins, K. R.
The TFTP Protocol.
DARPA Internet Network Working Group, Request for Comments #783, June, 1981.

[Sun 84] Sun Microsystems.
Engineering Manual for the Sun-2/120 CPU Board.
Technical Report 800-1185-01, Sun Microsystems, Inc., September, 1984.