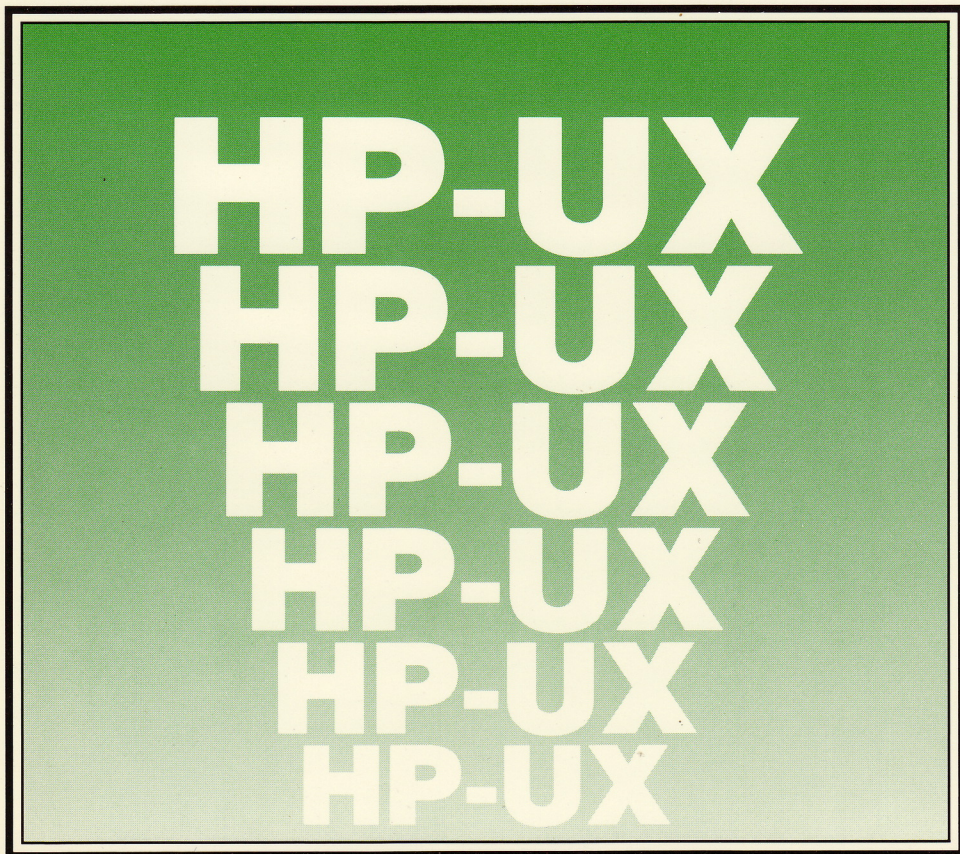


HP-UX Concepts and Tutorials
Vol. 3: Software Development Tools



HP-UX Concepts and Tutorials Vol. 3: Software Development Tools

Manual Reorder No. 97089-90040

© Copyright 1985 Hewlett-Packard Company

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

© Copyright 1980, Bell Telephone Laboratories, Inc.

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1984...First Edition - Part numbered 97089-90004 was 4 volumes and was shipped with HP-UX 4.0 on Series 500 Computers and with HP-UX 2.1, 2.2, 2.3, and 2.4 on Series 200 Computers. Each volume did not have an individual part number. This was obsoleted in April, 1985 and replaced with Manual Kit #97070-87903 which includes:

Title	Manual P/N	Binder P/N
Vol. 1: Text Processing and Formatting	97089-90020	9282-1023
Vol. 2: Programming Environment	97089-90030	9282-1023
Vol. 3: Software Development Tools	97089-90040	9282-1023
Vol. 4: Shells and Miscellaneous Tools	97089-90050	9282-1023
Vol. 5: Data Communications	97089-90060	9282-1023
Vol. 6: Graphics	97089-90070	9282-1023

April 1985...Edition 1 - Volume 3: Software Development Tools

Contents

The articles contained in *HP-UX Concepts and Tutorials* are provided to help you use the commands and utilities provided with HP-UX. The articles have several sources. Some were written at Hewlett-Packard specifically for HP computers. Others were written at Bell Laboratories or University of California at Berkeley and have been tailored for HP computers.

HP-UX Concepts and Tutorials has six volumes:

- Volume 1: Text Processing and Formatting
- Volume 2: Programming Environment
- Volume 3: Software Development Tools
- Volume 4: Shells and Miscellaneous Tools
- Volume 5: Data Communications
- Volume 6: Graphics

This is “Vol. 3: Software Development Tools” and the articles it includes are:

1. Make: A Program for Maintaining Computer Programs
2. SCCS User's Guide
3. Device I/O Library
4. Lex: A Lexical Analyzer Generator
5. Yacc: Yet Another Compiler-Compiler
6. The ADB Debugger
7. The CDB Debugger

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard computer system products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of shipment.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

HP 9000 Series 200

For the HP 9000 Series 200 family, the following special requirements apply. The Model 216 computer comes with a 90-day, Return-to-HP warranty during which time HP will repair your Model 216, however, the computer must be shipped to an HP Repair Center.

All other Series 200 computers come with a 90-Day On-Site warranty during which time HP will travel to your site and repair any defects. The following minimum configuration of equipment is necessary to run the appropriate HP diagnostic programs: 1) .5 Mbyte RAM; 2) HP-compatible 3.5" or 5.25" disc drive for loading system functional tests, or a system install device for HP-UX installations; 3) system console consisting of a keyboard and video display to allow interaction with the CPU and to report the results of the diagnostics.

To order or to obtain additional information on HP support services and service contracts, call the HP Support Services Telemarketing Center at (800) 835-4747 or your local HP Sales and Support office.

*For other countries, contact your local Sales and Support Office to determine warranty terms.

Table of Contents

Make: A Program for Maintaining Computer Programs

Introduction	1
Basic Features	2
Description Files and Substitutions	4
Command Usage	6
Implicit Rules	7
Example	8
Suggestions and Warnings	10
Appendix: Suffixes and Transformation Rules	11



Make: A Program for Maintaining Computer Programs

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (such as *Yacc* or *Lex*). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command *make* is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last “*make*”. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think → edit → *make* → test ...

Make runs on the HP-UX operating system, and is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple-source versions or of describing huge programs.

Basic Features

The basic operation of *Make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *IS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line:

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
      cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information were stored in a file named *Makefile*, the command:

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three ".o" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. >From the file system, *make* discovers that there are three ".c" files corresponding to the needed ".o" files, and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make's* innate knowledge:

```
prog : x.o y.o z.o
      cc x.o y.o z.o -lS -o prog

x.o : x.c defs
      cc -c x.c

y.o : y.c defs
      cc -c y.c

z.o : z.c
      cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new “.o” files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry “save” might be included to copy a certain set of files, or an entry “cleanup” might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
${xy}
$Z
$(Z)
```

The last two invocations are identical. \$\$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: \$*, \$@, \$?, and \$<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -ls
prog: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o prog
...
```

The command

```
make
```

loads the three object files with the *IS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex (“-ll”) and the Standard (“-lS”) libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in HP-UX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
Z = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns *LIBES* the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2 . . .] : [ : ] [dependent1 . . .] [; commands] [# . . .]
[(tab) commands] [# . . .]
. . .
```

Items inside brackets can be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters “*” and “?” are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the “-i” flag has been specified on the *make* command line, if the fake target name “.IGNORE” appears in the description file, or if the command string in the description file begins with a hyphen. Some HP-UX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set.

- \$@ is set to the name of the file to be “made”.
- \$? is set to the string of names that were found to be younger than the target.

If the command was generated by an implicit rule (see below),

- \$< is the name of the related file that caused the action, and
- \$* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name “.DEFAULT” are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
make [ flags ] [ macro definitions ] [ targets ]
```

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

- i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.
- s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.
- r Do not use the built-in rules.
- n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.
- t Touch the target files (causing them to be up to date) rather than issue the usual commands.
- q Question. The .IT make command returns a zero or non-zero status code depending on whether the target file is or is not up to date.
- p Print out the complete set of macro definitions and target descriptions
- d Debug mode. Print out detailed information on files and times examined.
- f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

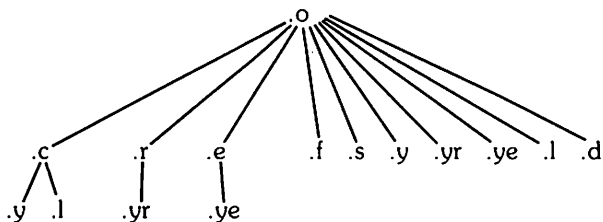
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (Descriptions of these tables and means of overriding them are included at the end of this tutorial.) The default suffix list is:

`.o` Object file
`.c` C source file
`.e` Efl source file
`.r` Ratfor source file
`.f` Fortran source file
`.s` Assembler source file
`.y` Yacc-C source grammar
`.yr` Yacc-Ratfor source grammar
`.ye` Yacc-Efl source grammar
`.l` Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros `AS`, `CC`, `RC`, `EC`, `YACC`, `YACCR`, `YACCE`, and `LEX`. The command

```
make CC=newcc
```

causes the “newcc” command to be used instead of the usual C compiler. The macros `CFLAGS`, `RFLAGS`, `EFLAGS`, `YFLAGS`, and `LFLAGS` may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command
```

```
P = und -3 opr -r2      # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c gram.y lex.c gcoc.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES = -IS
LINT = lint -p
CFLAGS = -O
```

```
make:    $(OBJECTS)
         cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
         size make
```

```
$(OBJECTS): defs
gram.o: lex.c
```

```
cleanup: -rm *.o gram.c
         -du
```

```
install:@size make /usr/bin/make
cp make /usr/bin/make ; rm make
```

```
print:   $(FILES)      # print recently changed files
         pr $? $P
         touch print
```

```
test:    make -dp grep -v TIME >1zap
         /usr/bin/make -dp grep -v TIME >2zap
         diff 1zap 2zap
         rm 1zap 2zap
```

```
lint:    dosys.c doname.c files.c main.c misc.c version.c gram.c
         $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
         rm gram.c
```

```
arch:    ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -ls -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the “size make” command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The “print” entry prints only the files that have been changed since the last “make print” command. A zero-length file *print* is maintained to keep track of the time of the printing; the \$? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
```

or

```
make print "P= cat >zap"
```


Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has an *#include "defs"* line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the “-n” option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the “-t” (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

(“touch silently”) causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag (“-d”) causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the “-r” flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name “.SUFFIXES”; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a “.r” file to a “.o” file is thus “.r.o”. If the rule is present and no explicit command sequence has been given in the user’s description files, the command sequence for the rule “.r.o” is used. If a command is generated by using one of these suffixing rules, the macro \$* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro \$< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for “.SUFFIXES” in his own description file; the dependents will be added to the usual list. A “.SUFFIXES” line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    (CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    (EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    (AS) -o $@ $<
.y.o :
    (YACC) $(YFLAGS) $<
    (CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    (YACC) $(YFLAGS) $<
    mv y.tab.c $@
```


Table of Contents

SCCS User's Guide	1
Introduction	2
Learning the Lingo	2
S-Files	2
Deltas	2
SID's (Version Numbers)	2
ID Keywords	3
Creating SCCS Files	4
Removing SCCS Files	5
Getting Files for Compilation	5
Changing Files (Creating Deltas)	6
Getting a Copy to Edit	6
Merging the Changes Back Into the S-File	6
When To Make Deltas	7
What's Going On: The Sact Command	7
Using ID Keywords	7
Creating New Releases	9
Cancelling an Editing Session	9
Restoring Old Versions	11
Reverting to Old Versions	11
Selectively Excluding Old Deltas	11
Selectively Including Deltas	12
Removing Deltas	13
The Help Command	13
Auditing Changes	14
The Prs Command	14
Determining Why Lines Were Inserted	15
Comparing Versions	15
Files Used By SCCS	16
S-Files	16
G-Files	17
L-Files	17
P-Files	18
D-Files	18
Q-Files	18
X-Files	19
Z-Files	19
Concurrent Editing	19
Concurrent Edits on Different Versions	19
Concurrent Edits on the Same Version	20

Saving Yourself	20
Making Temporary Changes	20
Recovering an Edit File	20
Restoring the S-File	21
Using the Admin Command	22
Creating SCCS Files	22
Adding Comments to Initial Delta	22
Descriptive Text in Files	22
Setting SCCS File Flags	23
Specifying Who Can Edit a File	24
Maintaining Different Branches	26
Creating a Branch	26
Retrieving a Branch	26
Branch Numbering	26
A Warning	27
SCCS's Protection Facilities	28
General File Protection	28
System Protection Using Admin	29
Using SCCS With Make	29
To Maintain Groups of Programs	30
To Maintain a Library	31
To Maintain a Large Program	32
Using SCCS on a Multi-User Project	33
How the SCCS Interface Works	33
Configuring an SCCS System Using the Interface	34
Quick Reference	37
Commands	37
ID Keywords	39

SCCS User's Guide

Introduction

SCCS (Source Code Control System) is simply a set of HP-UX commands which allow you to:

- track all changes made to a text file;
- retrieve the current (latest) version of a file;
- retrieve any previous version of a file, ignoring any changes made to the original after a given revision;
- control who changes a file;
- keep track of the date and location of each change made to a file along with the name of the person making the change;
- add comments when each change is made indicating the reason for that change.

One application of SCCS is to keep track of source files during the development and maintenance of large systems. This article is directed towards this use of SCCS; however, it can be used in any project that involves supporting groups of related text files. Object code cannot be maintained under SCCS.

Once you store a program's source file under SCCS, all of its versions, plus additional log information, are kept in a file called the "s-file". S-files are also referred to as "SCCS files" and must have a "s." prefix on their name. There are three major operations you can perform on the s-file:

1. Get a file for some non-editing purpose, such as compilation. This operation retrieves a version of the file from the s-file that is read-only. By default, the latest version of the file is retrieved. This file is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.
2. Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a particular version of an s-file at a time (unless you have specifically allowed concurrent edits on the same version).
3. Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

Learning the Lingo

There are a number of terms that are worth learning before using SCCS.

S-files

An s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made. A description of what this header information includes is presented later in this article.

Deltas

Each set of changes to the s-file (which is approximately equivalent to a version of the file) is called a delta. Although technically a delta only includes the changes made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before. This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes. All of the deltas of a file maintained under SCCS are stored in an s-file.

SID's (Version Numbers)

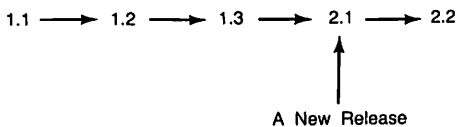
A SID (SCCS ID) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. The form of two-part SIDs is:

```
release.level
```

where "release" and "level" are non-zero, positive integers. Normally the release number stays the same while the "level" increments with each delta. However, you can move into a new release of a file if some major change is being made. Since all past deltas are normally applied when version is retrieved, the SID of the final delta applied is used to represent the version number of the file as a whole.

Deltas applied to one SCCS file can be considered nodes of a tree, the initial version of the file being the root node. The root delta (node) normally has the SID number "1.1" and the deltas that follow are "1.2", "1.3", etc. The naming of successor deltas by incrementing the SID level number is performed automatically by SCCS when you retrieve a file for editing with *get -e*, although the delta itself is not created until you execute *delta*.

The diagram below illustrates the development of an SCCS file where each delta depends on all of the previous deltas.



ID Keywords

When you retrieve a version of a file from SCCS with intent to compile it (or rather, do something other than edit it), some special keywords are expanded by SCCS when they are found in the file. These ID keywords can be used to include the current version number or other information into the file. All ID keywords are of the form %x%, where “x” is an upper case letter. For example, %I% is the SID of the latest delta applied in retrieving a particular version, %W% includes the module name, SID, and a string of characters that makes it findable by the *what* command, and %G% is the date of the latest delta applied. A list of all of the ID keywords can be found in the Quick Reference section at the end of this article and in the entry for *get* in the *HP-UX Reference*.

For example, assume that you have a source file stored under SCCS and it contains the line of code:

```
static char SccsId[] = "%W%";
```

When you retrieve the file for editing, the text file will contain the line just as it appears above. However, when you retrieve the file for compilation the %W% is expanded to indicate the module name, SID, and the string of characters recognized by *what*:

```
static char SccsId[] = "@(#)prog.c 1.2 05/15/84";
```

The *what* command is a valuable tool for quickly finding out information about a particular version of a program. To use it the program’s source code must be contained in SCCS files. In the SCCS files, any string of information that you want to be accessed by *what* must begin with the ID keyword %Z%. (%W%, mentioned earlier, is actually a combination of several ID keywords, including %Z%.) When the files are retrieved for compilation, this ID keyword is expanded to the string: @(#). When you invoke *what* on a file, the command prints out anything it finds between this string and the first “, >, \, newline, or null character. Refer to the section “Using ID Keywords” for more information about *what*.

When you retrieve a file for editing, the ID keywords are not expanded; this is so that after you store the file back into SCCS, they can still be expanded automatically when the file is retrieved for compilation. If you edit and store a version of a file in which the ID keywords are expanded, SCCS can no longer control the updating of the ID keywords’ values. For example, if you use the ID keyword for the file’s version and then store the keyword’s expanded value, all of the following versions will indicate that same version number – SCCS can not increment it. Also, if you compile a version of the program without expanding a version number ID keyword that appears in it, it is impossible to tell what version it is since all that the code will contain is “%I%”.

Creating SCCS Files

To put source files into SCCS format, use the *admin* command. The following stores a file called "s.file" under SCCS:

```
admin -ifile s.file
```

The *-i* keyletter indicates that *admin* is to create a new SCCS file (called an s-file) and "initialize" its contents with the contents of the file "file". The "s.file" argument is the name of the s-file. **All s-file names must begin with "s."** The initial version of s.file is a set of changes (delta 1.1) applied to a null s-file.

After creating a new s-file, *admin* returns the message:

```
No id keywords (cm7)
```

if you have not included any ID keywords in it. This is just a warning message and it is discussed further in a later section.

Since you have stored the contents of "s.file" under SCCS, you can now remove the original file:

```
rm file
```

Note that if the name of the SCCS file is the same as the original text file except for the "s." prefix, then original file must be removed or moved to another directory. This is because when you retrieve a version of an SCCS file, the name of the resulting text file is the SCCS file name with the "s." removed. If there is already a writeable file with this name in your current directory, SCCS does not allow you to retrieve the SCCS file version in most cases.

Assume that your current HP-UX directory contains several C source files that you want to maintain under SCCS. The following shell script stores each under SCCS with the required "s." prefix added onto its name and removes the original source files.

```
for i in *.c
do
    admin -i$i s.$i
    rm $i
done
```

If you want to have ID keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* prints "No Id Keywords (cm7)" after each s-file is created. If you create an s-file without ID keywords and then later decide to add them, merely retrieve the file for editing, add the ID keywords, store the changes, and then state that ID keywords have been added when you are prompted for comments.

Removing SCCS Files

In order to protect s-files, SCCS does not supply a direct method of removing them from your system. S-files are protected from accidental deletion in two ways:

- They are created as read-only files.
- There is no SCCS command that removes them.

Because of this protection, you must make the files writeable before you can remove them. Use *chmod* to change the access permission on an s-file:

```
chmod +w s.file
```

The “+w” indicates that you are adding write access to the file “s.file”. Once you have a writeable s-file, you can remove it with:

```
rm s.file
```

Getting Files for Compilation

To get a copy of the latest version of the SCCS file “s.file”, type:

```
get s.file
```

Get responds, for example, with:

```
1.1  
87 lines
```

indicating that version 1.1 was retrieved and that it has 87 lines. The retrieved text is placed in a file in the current directory whose name is formed by deleting the “s.” prefix. The file is read-only to remind you that you are not supposed to change it. If you do make changes, they are lost the next time someone does a *get*.

To retrieve all of the SCCS files in a directory so that they can be compiled, specify the directory name as an argument to *get*:

```
get directory
```

The retrieved text files are placed in your current directory and any non-SCCS files (files without the “s.” prefix) in the directory are silently ignored.

Note that if the s-file (or the directory containing s-files) that you want to access is not located in your current directory you must specify its full pathname.

Changing Files (Creating Deltas)

Getting a Copy to Edit

To edit a source file, you must first use *get* with its *-e* (*e* for edit) keyletter to retrieve it:

```
get -e s.file
```

Get responds:

```
1.1
87 lines
New delta 1.2
```

The retrieved file “file” (without the “s.” prefix) is placed in your current directory and you have read and write access to it. Edit the file using a standard text editor, for example *vi*:

```
vi file
```

To retrieve all of the SCCS files in a directory for editing, specify the directory name as an argument to *get -e*:

```
get -e directory
```

Merging the Changes Back Into the S-File

When the desired changes have been made to the text file, you can store the changes in the SCCS file using the *delta* command:

```
delta s.file
```


assuming that the s-file is located in your current directory. If it is located in a different directory you must specify a pathname for the s-file. *Delta* prompts you for “Comments?” before it merges the changes in. At this time you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash \). *Delta* then responds, for example, with:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged. (Changes to a line are counted as a line deleted and a line inserted.) Finally, SCCS removes “file” from your current directory; you can retrieve it again using *get*.

Note that the comments that you are prompted for are not maintained as part of the text body of the s-file, but are kept in another section of the s-file that is used internally by SCCS.

When To Make Deltas




It is probably unwise to make a delta before every recompilation or test, unless other people may need to edit the file at the same time. Creating too many deltas may result in unclear comments, such as “fixed compilation problem in previous delta” or “fixed botch in 1.3”. However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, *delta* everything being edited, *re-get* them, and recompile everything.

Working on a project with several people presents a problem when two people need to modify a particular version of a file at the same time. SCCS prevents this by locking the version while it is being edited (unless concurrent editing of one version has been specifically allowed). This means that you should not retrieve a file for editing unless you are actually going to edit it at the time, since you will be preventing other people on the project from making necessary changes. As a general rule, all source files that you are editing should be stored with *delta* before being used in compilations. This gives other users a better chance of being able to edit files when they need to.

What’s Going On: The *sact* Command

To find out who is currently editing an SCCS file, use:

```
sact s.file
```



For each editing session taking place on the file, *sact* (SCCS activity) tells you which SID (version) is being edited, what SID will be assigned to the new delta when editing is done, who is doing the editing, and the data and time that editing began (when *get -e* was invoked). If no one is editing “s.file”, *sact* returns an error message telling you that a p-file does not exist for the file (the “Types of Files” section later in this tutorial discusses p-files).

You can specify more than one SCCS file name as arguments to *sact*; each file is checked one at a time. You can also specify a directory, in which case *sact* checks every SCCS file in that directory and silently ignores non-SCCS files (files without the “s.” prefix).


Using ID Keywords

ID keywords inserted into your file are expanded when you retrieve a file for compilation with *get*. They record information about the file, such as the time and date it was created, the version retrieved, and the module’s name. For example, a line in an SCCS file such as:

```
static char ScCsId[] = "ZWZ\tZGZ";
```

is replaced with something like:

```
static char ScCsId[] = "@(#)prog.c      1.2      08/29/80";
```



in the retrieved source file. This tells you the name and version of the source file and the time the delta was created. The string “@(#)” is the expanded form of the keyword %Z% and is searched for by the *what* command. (Note that the %W% ID keyword shown above is shorthand for several other ID keywords including %Z%.) It makes it possible to quickly locate expanded ID keywords in text files using *what*. Note that when you retrieve a file for editing the keywords are not expanded. This is so that they will still be in their original form when you store the file again with *delta*.

Approximately 20 ID keywords are available for you to use in your SCCS files. The “Quick Reference” section at the end of this tutorial contains a list of them and a list can also be found under the entry for *get* in the *HP-UX Reference*.

The What Command

When %Z% is used, expanded ID keywords in files can be located using *what*. To find out the current version number of a source file and what version of it is used in an object file and final program (assuming you have previously inserted the necessary ID keywords in the SCCS source file), use:

```
what file.c file.o a.out
```

What prints all strings it finds that begin with “@(#)” in the three files. It works on all types of files, including binaries and libraries. For example, the above command outputs something like:

```
file.c:
  file.c 1.2      08/29/80
file.o:
  file.c 1.1      02/05/79
a.out:
  file.c 1.1      02/05/79
```

From this you see that the source in “file.c” does not compile into the same version as the binary in “file.o” and “a.out”.

What searches the given files for all occurrences of the string “@(#)”, which is the replacement for the %Z% ID keyword, and prints what follows that string until the first double quote (“), greater than (>), backslash (\), newline, or (nonprinting) NUL character. Note that you can locate and display constant text as well as ID keywords with *what* if you precede that text with %Z%.

For example, assume an SCCS file “s.prog.c” contains the following line:

```
char id[] "ZZZMZ:ZIZ;
```

Note that the colon (“:”) is not part of an ID keyword. It is left unchanged when the ID keywords are expanded. Next, the command line:

```
set s.prog.c
```

is executed. The retrieved file “prog.c” is then compiled to produce “prog.o” and “a.out”. The command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
  prog.c:1.2
prog.o:
  prog.c:1.2
a.out:
  prog.c:1.2
```

indicating that version 1.2 of the file “prog.c” was used in all three files.

Where to Put Id Keywords

ID keywords can be inserted anywhere in SCCS files, including comments. ID keywords that are compiled into the object module are especially useful, since they let you compare what version of the object is being run to the current version of the source.

When you put ID keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W% %GX";
```

in the file "access.h" and:

```
static char OpsysSid[] = "%W% %GX";
```

in the file "opsys.h". If you used the same variable name in both, you get compilation errors because the variable is redefined. You should also be aware that if you place ID keywords in a header file as code that is eventually compiled and then included that header file in several modules that are loaded together, the same version information will appear several times in the resulting object module. A solution is to insert header file's ID keywords as comments.

Creating New Releases

When you want to create a new release of a program, you can specify the new release number using *get*'s *-r* keyletter. For example:

```
get -e -r2 s.prog.c
```

retrieves the release 1's latest version of "s.prog.c" and causes the next delta to be in release 2 (an SID of 2.1). Future deltas are automatically in release 2.

To assign a new release number for all of the SCCS files in a directory, use:

```
get -e -r2 directory
```

assuming that the previous release was release 1, and then execute:

```
delta directory
```

All of the SCCS files in the directory are assigned a new delta SID of 2.1

Cancelling an Editing Session

If you retrieve a file for editing with *get -e* and then decide that you do not want to edit it, cancel the editing session with:

```
unset s.file
```

Unget returns the SID of the cancelled delta. Only the person who began an editing session can cancel it. *Unget* can accept more than one file name argument or, alternatively, use:

```
unset -
```

in which case *unget* accepts file names from standard input.

If you are currently editing a number of SCCS files in one directory and want to cancel all of the editing sessions for them, you can specify the directory:

```
unget directory
```

In this case *unget* checks every SCCS file in the directory. If one of the files is not currently being edited, *unget* returns an error message indicating that its associated p-file does not exist (see “Files Used by SCCS” section later in this tutorial).

If you are currently editing more than one version of a file, *unget*'s -r keyletter allows you to specify which version's editing session you want to cancel:

```
unget -r2.3 s.file
```

If you find that you retrieved a file for editing when actually you needed for some other purpose, you would like to cancel the editing session but keep the file in the current directory. Normally when you cancel an editing session, *unget* removes the retrieved text file from the current directory. You can request that it not be removed with the -n keyletter:

```
unget -n s.file
```

This leaves the text file “file” still available for inspection or compilation, but any changes made to the file cannot be stored back in the SCCS file with *delta*.

You can request that *unget* execute silently (not print out the file's cancelled delta's SID) using the command's -s keyletter:

```
unget -s s.file
```

Restoring Old Versions

This section discusses how *get*'s *-r*, *-x*, and *-i* keyletters are used to retrieve various versions of a file. They can be used in any combination. The *-e* keyletter can also be used with them to create a new delta based on particular versions.

Reverting to Old Versions

Normally, *get* retrieves the latest version of the specified file. However, you can request a particular version using *get*'s *-r* keyletter.

Suppose that after delta 1.2 was stable you made and released a delta 1.3. However, this introduced a bug, so you made a delta 1.4 to correct it. Then you found that 1.4 was still buggy, and you decided you wanted to go back to the old version. You can access delta 1.2 by choosing the SID in a *get*:

```
get -r1.2 s.Prog.c
```

This produces a version of "prog.c" that is delta 1.2. Any changes that you made between delta 1.2 and the most recent delta are ignored.

If you specify a release number but not a level number, the highest level number that exists within that release is retrieved. *Get -r* also allows you to retrieve particular branch deltas. Branches are discussed in the section "Maintaining Different Branches" later in this article.

If you try to retrieve for compilation a particular version that does not exist, SCCS responds with an error message. There is one exception: if you specify only a release number and that release doesn't exist, SCCS retrieves the delta with the highest release number that does exist, and with the highest level number within that release.

In some cases you don't know what the SID of the delta you want is. However, *get* allows you to revert to the version of the program that was running as of a certain date using its *-c* (cutoff) keyletter. For example,

```
get -c840722120000 Prog.c
```

retrieves whatever version was current as of July 22, 1984 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line is equivalently stated with:

```
get -c"84/07/22 12:00:00" Prog.c
```

Selectively Excluding Old Deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this with the *-x* keyletter:

```
get -e -x1.3 s.Prog.c
```


When delta 1.5 is made, it includes the changes made in delta 1.4, but excludes the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you don't want to include 1.3 and 1.4 you can use:

```
get -e -x1.3-1.4 s.Prog.c
```

which excludes all deltas from 1.3 to 1.4. Alternatively,

```
get -e -x1.3-1 Prog.c
```

excludes a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using the `-x` keyletter (or `-i`, see below) there are conflicts between versions. For instance, it may be necessary to both include and delete a particular line, in which case SCCS always prints out a message telling the range of lines affected; these lines should then be examined very carefully to see if the version SCCS got is correct.

Since each delta (in the sense of "a set of changes") can be excluded at will, it is usually useful to put each semantically or conceptually distinct change into its own delta.

Selectively Including Deltas

Just as `get's -x` keyletter allows you to exclude deltas from a version in which they are normally included, the `-i` allows you to include deltas that are not normally included.

For example, assume that you have an SCCS file containing five deltas, 1.1 through 1.5. To retrieve a version of a file containing only deltas 1.1, 1.3, and 1.5, request that version 1.1 be retrieved and force the inclusion of deltas 1.3 and 1.5:

```
get -r1.1 -i1.3,1.5 s.file
```

To retrieve version 1.5 all of the deltas must be used. All of the following `get` command lines accomplish this.

```
get -r1.5 -i1.2 s.file
```

```
get -r1.5 s.file
```

```
get s.file
```

Note that the `-i` keyletter in the first command line has no effect since delta 1.2 is already used to construct version 1.5. The `-r` keyletter is not required either since delta 1.5 is the most recent delta and, by default, `get` retrieves the version incorporating it.

If there are conflicts between versions when you use the `-i` keyletter, SCCS provides a message indicating the range of lines affected, just as it does when the `-x` keyletter is used. You should examine these lines in the retrieved file to make sure that they are correct.

Removing Deltas

Get -x allows you to exclude deltas from the retrieved file; however, the deltas are not removed from the SCCS file and the information they contain is still available and consuming space. To permanently remove a delta from an SCCS file, use *rmDEL*. *RmDEL* requires that you use the *-r* keyletter to specify which delta is removed:

```
rmDEL -r1.3 s.file
```

Before you can use *rmDEL* to remove a delta, all of the following requirements must be met:

- the specified version of the file is not currently being edited;
- the SID must be the most recent delta on its branch of the delta chain for the named file: no other deltas can depend on it;
- you originally created the delta or you are the owner of the SCCS file and the directory that it is in.

The Help Command

Error messages returned by the SCCS commands have the form:

```
ERROR : message (code)
```

If it is not clear from “message” why the error occurred, use the associated “code” as an argument to the *help* command. Invoking:

```
help code
```

often provides a little more explanation about the cause of the error. For example, if you execute “get program” you could receive the following message:

```
ERROR[Program]: not an SCCS file (col)
```

Executing:

```
help col
```

produces:

```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s."
```

Auditing Changes

The Prs Command

When you create deltas, you presumably give reasons for the deltas in response to the “comments?” prompt. To print out these comments later, use:

```
prs s.file
```

Note that *prs* provides information about each of the deltas used to create the requested version of the file; therefore, it is a way to list the deltas upon which a particular version depends. It produces a report for each delta providing the time and date of creation, the user who created the delta, and the comments associated with the delta. For example, the output of the above command might be:

```
s.file:

D 1.3 84/04/12 08:21:35 becky 3 2 00020/00008/00021
MRs:
COMMENTS:
inserted 20 lines, removed 8 lines

D 1.2 84/04/11 09:21:08 becky 2 1 00008/00000/00021
MRs:
COMMENTS:
inserted 8 lines

D 1.1 84/04/10 06:37:14 becky 1 0 00021/00000/00000
MRs:
COMMENTS:
date and time created 84/04/10 06:37:14 by becky
```

The report indicates that the file’s initial delta (created with *admin -i*) inserted 21 lines, delta 1.2 inserted 8 lines and left 21 unchanged, and delta 1.3 inserted 20 lines, removed 8 lines, and left 21 lines unchanged.

You can request information about a particular version of a file using *prs*’s *-r* keyletter:

```
PRS -r2.3 s.PROG.C
```

Prs can accept multiple file names or directory names as arguments. If you request information about all of the SCCS files in a directory, you should probably redirect *prs*’s output to a file and look at it at your leisure:

```
PRS directory >output
```

When a directory is specified, the effect is as if each SCCS file it contains were named and any non-SCCS files are ignored.

Prs also allows you to modify the information it provides using its *-d* keyletter. Refer to the *prs* entry in the *HP-UX Reference* to see how this is done.

Determining Why Lines Were Inserted

To find out why you inserted various lines in a file, you can get a copy of the file with each line preceded by the SID of the delta that created it using:

```
get -m s.PROG.C
```

where the retrieved copy is called "prog.c". Once you have determined which delta inserted the line you are interested in, use *prs* to find out what that particular delta did by looking at its comment line.

Another way to find out which lines were inserted by a particular delta (e.g., 1.3) is:

```
get -m -p s.PROG.C | grep '^1.3'
```

The *-p* flag causes *get* to output the retrieved text to the standard output rather than to a file.

Comparing Versions

To compare two versions of a file, use *scsdiff*. For example,

```
scsdiff -r1.3 -r1.6 s.PROG.C
```

outputs the differences between delta 1.3 and delta 1.6 in a format similar to the format used by the *diff* command.

You can specify any number of file names with *scsdiff* but the same two SID's specify which versions are compared for all of them. You can not specify a directory as an argument.

Files Used by SCCS

As a user of SCCS, you do not need to know all of the information covered in this section; however, it should give you a feel for the inner workings of SCCS.

There are 8 types of files that are used by SCCS and all of them are ASCII text files. They are:

S-files	SCCS files created by <i>admin -i</i> .
G-files	Text files containing the “body” of SCCS files and created by <i>get</i> .
L-files	Files containing delta dependency information and created by <i>get -l</i> .
P-files	Files created and used by SCCS to keep track of multiple edits.
D-files	Temporary files created and used by SCCS during the execution of <i>delta</i> .
Q-files	Temporary files created and used by SCCS to update p-files.
X-files	Temporary files created and used by SCCS to update s-files.
Z-files	Lock-files created and used by SCCS to prohibit simultaneous updating of s-files.

Normally, only 4 of these file types are visible to users of SCCS: s-files, g-files, l-files, and p-files. The remaining 4 types are temporary files used internally by SCCS during the execution of particular commands.

S-Files

S-files are often referred to as SCCS files in this tutorial. They contain all of the versions of files you are maintaining under SCCS. You create and name an s-file when you initially enter a file into SCCS:

```
admin -ifile s.file
```

“s.file” is the new s-file and “file” can now be removed. Accessing a file maintained under SCCS using SCCS commands is done using its s-file name. S-file names must begin with the prefix “s.”.

The Contents of the S-File

S-files are composed of lines of ASCII text arranged in the following 6 parts:

Checksum	A line containing the “logical” sum of all the characters of the file, not including the checksum itself.
Delta Table	Information about each delta, such as type, SID, data and time of creation, and user inserted comments.
User Names	A list of login names and/or group IDs of users who are allowed to modify the file by adding or deleting deltas. You modify it using <i>admin</i> .
Flags	Indicators that control certain actions of various SCCS commands. You modify them using <i>admin</i> .
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file. You modify it using <i>admin</i> .
Body	The actual text that is being administered by SCCS, intermixed with internal SCCS control lines. You modify it using <i>get -e</i> and <i>delta</i> .

You modify the Body section of the s-file whenever you create or delete deltas. You modify the User Names, Flags, and Descriptive Text sections using the *admin* command (see the “Using Admin” section later in this article). The Checksum and Delta Table are modified internally by SCCS.

Since the entire contents of an s-file is ASCII, the file can be processed with various HP-UX commands, such as *vi*, *grep*, and *cat*. This is convenient but somewhat risky in those instances in which an SCCS file must be modified manually (e.g. when the time and date of a delta were recorded incorrectly because the system clock was set incorrectly) or when you simply want to look at its contents.

Note

If you modify the SCCS file directly (instead of with SCCS commands), the Checksum value may be incorrect, causing you to receive an error whenever you try to retrieve a version of the file. This problem is discussed in a later section, “Restoring the S-File”. You should not edit an s-file directly unless you completely understand its format.

G-Files

The *get* command creates a text file that contains a particular version of an s-file, obtained by applying deltas to the initial version. This text file is called a **g-file** and its name is formed by removing the SCCS file’s “s.” prefix. It is this file that you use for inspection, compilation, or editing purposes.

G-files are created in the current directory and are owned by the real user. Their file mode depends on how *get* is invoked. If you use:

```
get s.file
```

the resulting g-file “file” has mode 444 (read only) and is produced for inspection or compilation, but not for editing. Note that any ID keywords in the file are expanded to their appropriate values.

If you use:

```
get -e s.file
```

then “file” can be edited. Note that any ID keywords in the file are not expanded, allowing them to be stored back in the file when you use *delta*.

L-Files

When you retrieve an SCCS file with *get*, you can request that an **l-file** be created using the command’s **-l** keyletter:

```
get -l s.file
```

The name of an l-file is formed by replacing the “s.” prefix of the SCCS file with “l.”. It contains a table indicating what deltas were used to create the retrieved version of an SCCS file. You must specifically request the creation of l-files with **-l**; by default *get* does not create them.

To send delta dependency information to standard output instead of placing it in an l-file, use:

```
get -r2.3 -lp s.file
```

P-Files

When you retrieve an SCCS file for editing (*get -e*), besides creating a writeable g-file containing the version's text, a **p-file** is also created. The name of a p-file is formed by replacing the "s." prefix of an SCCS file with "p."

P-files are used internally by SCCS to keep track of multiple edits on the same SCCS file (see "Concurrent Editing"). For each edit that is in progress on a particular SCCS file (*get -e* has been executed but not the associated *delta*), the file's p-file keeps track of:

- the SID of the retrieved version;
- the SID that will be given to the new delta when *delta* is executed;
- the login name of the user that executed *get -e*;
- the date and time that the *get -e* was executed.

If a p-file is accidentally destroyed, it can be regenerated with:

```
get -e -g s.file
```

The "-e -g" combination suppresses the retrieval of a writeable text file (g-file), but the associated p-file is created. A p-file must exist for an SCCS file before you can use *delta* on it.

When you request information with the *sact* command, you are presented with data from a p-file.

D-Files

D-files are used internally by SCCS during the execution of *delta* to hold a temporary copy of the original retrieved g-file before any editing was done. The name of a d-file is formed by replacing the "s." prefix of the associated SCCS file with "d.". When you retrieve an SCCS file for editing (*get -e*) and then invoke *delta*, SCCS creates a d-file and compares the edited g-file with the contents of the d-file to determine what has changed. These changes are then stored in the SCCS file (s-file).

When you invoke *delta*, you can request that the differences between the d-file and the g-file (the file that you retrieved and the file that you are now storing) be sent to standard output using:

```
delta -P s.file
```

Once *delta* is executed, you can request the same information with the *scsdiff* command.

Q-Files

A **q-file** is a temporary copy of a p-file that is used internally by SCCS. Its name is formed by replacing the "p." prefix of the p-file with "q.". Whenever a p-file needs to be updated (because editing of a version of a file was completed with *delta* or started with *get -e*), a q-file is first created. The change is made to the q-file and then the p-file is removed and the q-file is renamed to become the new p-file. This strategy is used to ensure the integrity of the p-file in case there are any problems adding or deleting entries from the table.

X-Files

An **x-file** is a temporary copy of an s-file that is used internally by SCCS. All SCCS commands that modify an SCCS file do so by first creating and modifying an x-file. This ensures that the SCCS file is not damaged if the processing terminates abnormally. The name of this temporary copy is formed by replacing the “s.” prefix of the SCCS file with “x.” When processing is complete, the old s-file is removed and the x-file is renamed to be the s-file.

Z-Files

Z-files are lock-files SCCS uses to prevent simultaneous updating of an SCCS file. They are discussed later in this article in the section “SCCS Protection Facilities”.

Concurrent Editing

Concurrent Edits on Different Versions

SCCS allows different versions of one SCCS file to be edited at the same time. This means that a number of *get -e* commands can be executed on the same file provided that no two executions retrieve the same version, unless concurrent edits on the same version are allowed (see the discussion in the next section).

SCCS uses a p-file to keep track of the edits that are in progress on one file. The first execution of *get -e* causes the creation of a p-file for the specified SCCS file. Subsequent executions of the command update the p-file, adding entries in the file for each edit session that is in progress. Each entry in the p-file specifies the SID of the retrieved version, the SID that will be assigned to the new delta, and the login name of the person doing the editing. When an editing session is terminated (with *delta* or *unget*), the corresponding entry in the file’s p-file is removed. If no other versions of the file are currently being edited, then the p-file itself is removed.

Before SCCS allows an editing session on a particular version of an SCCS file to begin, it makes sure that if a p-file for the file already exists there is no entry in it specifying that the version has already been retrieved. If there is no entry with that SID, SCCS adds an entry for the new editing session. If there is an entry with the same SID, SCCS generates an error message and does not allow the version to be retrieved for editing (unless multiple edits of the same version are allowed). SCCS informs you if editing is currently being done on another version of the file you request to edit.

Note

Multiple executions of *get -e* must be done from different directories. This is because each time any version of one file is retrieved, the resulting g-file (text file) is assigned the same name. As a result, SCCS prohibits multiple edits on the same file in the same directory because the g-file would constantly be overwritten.

In practice, multiple editing sessions are performed by different users with different working directories; therefore, this restriction normally does not cause a problem.

Concurrent Edits on the Same Version

By default, SCCS does not permit multiple executions of *get -e* on the same version of one SCCS file. Each editing session on a version begun with *get -e* must be ended with *delta* before another session can begin. However, you can change this and allow concurrent edits on the same version of a file by setting the file's *j* flag with the *admin* command (see the "Using Admin" section later in this article).

Note that if you do set a file's *j* flag, multiple editing sessions on the same version must be done in different directories, just like multiple edits on different versions.

Saving Yourself

Making Temporary Changes

If you use *get -e* to retrieve a file so that you can edit it, SCCS requires that you *delta* the changes that you make back into the associated *s*-file. Sometimes, however, it is necessary to make modifications to a file that you do not want saved.

To make temporary changes to a file possible, retrieve it from SCCS with:

```
get s.file
```

SCCS does not expect changes to be made to the file; therefore, it gives it read-only access. You must now change the mode of the file so that you can edit it:

```
chmod +w file
```

Chmod +w adds write access to a file. Any changes that you now make to "file" cannot be stored in SCCS.

Recovering an Edit File

Sometimes you may find that you have lost a file that you were trying to edit. Unfortunately, you can't just execute *get -e* again; SCCS keeps track of the fact that someone is trying to edit that version, so it won't let you do it again. Neither can you retrieve it using *get*, since that would expand the ID keywords. Instead, you can say:

```
get -k prog.c
```

This retrieves the file and does not expand the ID keywords, so it is safe to do a *delta* with it.

Restoring the S-File

You may find that the SCCS file itself is corrupt. The most common way this happens is when someone edits the file directly, not through the SCCS commands. SCCS keeps a checksum that contains the “logical” sum of all of the characters in the file. If you modify the SCCS file directly the checksum may have the wrong value. No SCCS command will process a corrupted SCCS file except *admin -h* and *admin -z* as described below.

You should audit all SCCS files for corruption on a regular basis. The simplest way to do this is to execute *admin* with its *-h* keyletter on all of the SCCS file:

```
admin -h s.file1 s.file2 ...
```

or:

```
admin -h directory
```

This checks to see if each file’s checksum is correct. The message “corrupted file (c06)” is produced for a file whose checksum is not correct.

If you have a corrupted SCCS file, you must first determine why its checksum is incorrect. If it is due to someone having directly modifying the file, the problem is often corrected by merely recomputing the checksum. Do this with *admin*’s *-z* keyletter:

```
admin -z prog.c
```

The checksum is recomputed to bring it into agreement with the actual contents of the file.

Note

Before you use *admin -z* you must find and correct the corruption problem. This is because once the checksum is recomputed, the corruption is no longer detectable. *Admin -z* does not find or fix the problem, it merely recomputes the checksum.

Using the Admin Command

Admin is used to create new SCCS files and change parameters of existing ones. When an SCCS file is created, its parameters are either initialized with keyletters or are assigned default values if no keyletters are specified.

Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file can use *admin* on it.

Creating SCCS Files

As discussed earlier, an SCCS file for a file called "prog" is created with:

```
admin -iprof s.prog
```

The name of the SCCS file is "s.prog". If no file name is specified with the *-i* keyletter, the text is read from standard input:

```
admin -i s.prog <prog
```

When the SCCS file is created, the release number assigned to its initial delta is normally "1" and the level number is always "1", meaning that the first delta of the file is "1.1". You can assign a different initial release number using *admin*'s *-r* keyletter when the file is created:

```
admin -iprof -r3 s.prog
```

Here, the initial delta is "3.1".

Adding Comments to Initial Delta

When you create an SCCS file, you can supply a comment stating the reason for the creation of the file. This is done with the *-y* keyletter:

```
admin -ifile -y"The reason this file was created" s.file
```

If you do not specify an initial comment with *-y*, SCCS gives the initial delta a comment line with the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

Descriptive Text in Files

A portion of an SCCS file is reserved for descriptive text, text that summarizes the content and purpose of the SCCS file. When you are creating an SCCS file you can insert descriptive text using *admin*'s *-t* keyletter followed by the name of a file containing the text:

```
admin -ifile -tdescrip s.file
```

You can either add descriptive text to an existing SCCS file or replace the descriptive text it already contains with:

```
admin -tnew_descrip s.file
```

where “new_descrip” is the name of the file containing the descriptive text. To remove descriptive text from an SCCS file, use `-t` without a file name:

```
admin -t s.file
```

To see the descriptive text for an SCCS file, use `prs` as follows:

```
prs -d:FD: s.file
```

`Prs`'s `-d` keyletter allows you to specify what information about the file that you want returned. The “:FD:” indicates that you want to see the file's descriptive text. Refer to the *HP-UX Reference* manual entry for `prs` for more information about the command's `-d` keyletter.

Setting SCCS File Flags

SCCS files have a number of parameters called **flags** that can be added and deleted using the `admin` command. These flags are maintained in a particular section of SCCS files along with their associated values where appropriate. Add flags with `admin`'s `-f` keyletter and delete them with its `-d` keyletter. For example:

```
admin -fd2.1 prog.c
```

sets the `d` flag to the value “2.1”. This flag can then be deleted using:

```
admin -dd prog.c
```

The flags that you can add with `admin -f` or delete with `admin -d` are:

- b** Allow branches to be made using `get -e -b`.
- dSID** Default SID to be used on a `get`. If this is just a release number, the default is the highest version number for that release.
- ceiling** Sets the highest release number for a file that can be retrieved with `get -e` to *ceiling*. *Ceiling* must be a number less than or equal to 9999. The default release ceiling for a file is 9999.
- ffloor** Sets the lowest release number for a file that can be retrieved with `get -e` to *ffloor*. *Floor* must be a number greater than 0 and less than 9999. The default release floor for a file is 1.
- i** Give a fatal error during `get` or `delta` if there are no ID keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the ID keywords inserted as constants instead of internal forms.
- j** Allow concurrent edits on the same version (SID) of the SCCS file.
- l***list* A *list* of releases that cannot be retrieved for editing (`get -e`). The *list* has the following syntax:

```
<list> ::= <range> | <list>,<range>
```

```
<range> ::= RELEASE_NUMBER | a
```

The character **a** is equivalent to specifying all of the releases for the names SCCS file. If you do not specify a *list* with the **l** flag, **a** is assumed by default.

To delete one or more “locked” releases with *admin*’s **-d** keyletter you must also use a *list* to specify which releases are to be “unlocked”. For example, “*admin -dla s.file*” unlocks all of the releases of *s.file* so that they can be edited.

- n** Causes *delta* to create a “null” delta in each of those releases (if any) being skipped when a delta is made in a new release (e.g. in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as “anchor points” so that branch deltas may later be created from them. If this flag is not set for a file, skipped releases are non-existent in the SCCS file, preventing branch deltas from being created from them in the future.
- qtext** Replace all occurrences of the ID keyword %Q% with the contents of file *text* when the SCCS file is retrieved for inspection or compilation. If the **q** flag has not been set for a file, occurrences of %Q% are not replaced with anything.
- mmodule** Replace all occurrences of the ID keyword %M% with the specified *module* name when the SCCS file is retrieved for inspection or compilation. If the **m** flag has not been set for a file, occurrences of %M% are replaced with the name of the SCCS file minus the “s.” prefix.
- ttype** Replace all occurrences of the ID keyword %Y% with the specified *type* when the SCCS file is retrieved for inspection or compilation. If the **t** flag has not been set for a file, occurrences of %Y% are not replaced with anything.
- v[pgm]** Causes *delta* to prompt for Modification Request (MR) numbers as the reason for creating a delta. If you set this flag when you create an SCCS file, *admin*’s **-m** keyletter must also be specified, even if its value is null.
- You can optionally specify an MR number validation checking program called “pgm” with *admin -fvpgm*.

Specifying Who Can Edit a File

Admin’s **-a** keyletter allows you to specify who can edit an SCCS file. Use it as follows:

```
admin -alogin s.file
```

where “login” is a user’s login name or an HP-UX group ID. If it is a group ID, the effect is equivalent to specifying all login names common to that group ID. Several **-a** keyletters may be used on a single *admin* command line.

Note that *admin* can accept one or more SCCS file names or directory names as arguments. For example, the command line:

```
admin -abill -ajane -ajohn directory
```

gives HP-UX users bill, jane, and john editing privileges to all of the SCCS files in the directory “directory”. The list of users for each SCCS file in the directory is updated to show this. No one else can edit the SCCS files there unless specifically given the right with *admin -a*.

If no one has been assigned editing privileges to a file with *admin -a*, the file's list of users is empty and anyone can edit the file (as long as they have write access to the file's directory).

A user's ability to edit an SCCS file is removed with *admin's -e* keyletter. For example,

```
admin -ebill directory
```

removes bill from the list of users allowed to edit the SCCS files in "directory".

Note

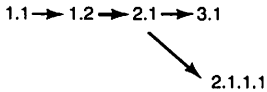
Before a user can be prohibited from editing a file, the file's list of users must be non-empty. If the list is empty everyone has editing privileges and using *admin -e* has no effect.

When a file's list of users is non-empty, any user not added to the list with *admin -a* is already prohibited from editing the file. Thus, you can remove a specific user's editing privileges only if you have previously added him to the list of users with *admin -e*.

Maintaining Different Branches

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a “branch.” Normally deltas continue in a straight line, each depending on the delta before. Creating a branch “forks off” a version of the program.

For example, in the diagram below there is one branch delta having an SID of 2.1.1.1:



The ability to create branches off of the latest main “trunk” delta must be enabled in advance by setting the file’s **b** flag:

```
admin -fb prog.c
```

The **b** flag can also be set when the SCCS file is first created. It is not necessary to set a file’s **b** flag in order to create a branch off of an older delta.

Creating a Branch

To create a branch off of the latest main trunk version, use:

```
set -e -b prog.c
```

If the retrieved version has an SID of 1.5 and no branch was previously created on it, a branch with SID 1.5.1.1 is created when the file is deltaed. The deltas for this branch are numbered 1.5.1.n where “n” increments by 1 with each delta.

If you retrieve an old version of an SCCS file for editing, SCCS automatically assigns a branch SID to the new delta. The file’s **b** flag need not be set to do this. For example, assuming that the latest delta of prog.c is delta 1.5 you can create a branch off of delta 1.2 using:

```
set -e -r1.2 prog.c
```

SCCS will automatically number the new branch delta 1.2.1.1 if it is the first branch off delta 1.2.

Retrieving a Branch

Deltas in a branch are not normally included when you use *get*. To retrieve these versions, you have to say:

```
set -r1.5.1 prog.c
```

specifying the requested branch’s SID.

Branch Numbering

SCCS uses the following SID numbering scheme for recognizing branch deltas:

```
release.level.branch.sequence
```

“Release.level” is the SID of the delta on the main trunk from which the branch descends. A “branch” number is assigned to each branch path that originates from a particular delta on the main trunk. A “sequence” number is assigned to each delta on a particular branch. Branch deltas always have all four of the above components in their SIDs and the release and level numbers are always those of the ancestral main trunk delta.

When you retrieve a branch, specifying only the release, level, and branch components of the SID returns the most recent version on a particular branch.

Although SCCS maintains enough internal information to remember delta dependencies of branch deltas, the SID number itself does not always indicate all of the deltas between a branch delta and its main trunk ancestor delta. For example, given delta 1.3.2.2 you know that the main path ancestor is delta 1.3 and that it is the second delta (sequence=2) on the second branch (branch=2) descending from delta 1.3. However, the diagrams below indicate two possible development paths for delta 1.3.2.2:

DIAGRAM 1:

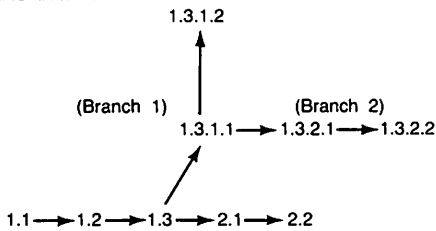
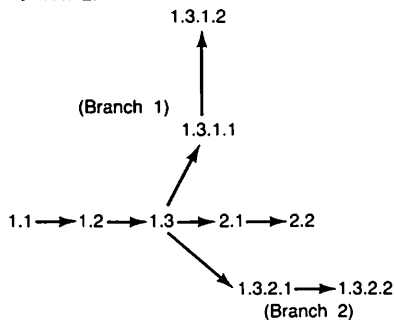


DIAGRAM 2:



Note that in Diagram 1, version 1.3.2.2 is dependent on deltas 1.1, 1.2, 1.3, 1.3.1.1, and 1.3.2.1, while in Diagram 2 the delta with the same SID is dependent on 1.1, 1.2, 1.3, and 1.3.2.1.

A Warning

Branches should be kept to a minimum. After the first branch from the main trunk, SID's are assigned rather haphazardly, and the structure gets complex very quickly.

SCCS's Protection Facilities

The protection facilities that SCCS provides for a system fall into two categories:

- general protection of files inherent to SCCS and that incorporates general HP-UX file system protection by appropriately setting the modes of various files;
- specific system protection strategies that you control with the *admin* command.

General File Protection

New SCCS files created with *admin* are given mode 444 (read only). This mode prevents any direct modification of the files by any non-SCCS commands. The mode of the files should not be changed to allow direct modification.

SCCS files must have only one link (name) because of the way SCCS modifies the files. Commands that modify SCCS files (*delta*, *admin*) create a copy of the file. The copy, called an x-file, is modified, the original SCCS file is removed, and the copy is renamed. If the original SCCS file has any links, they are broken when it is removed. SCCS generates an error message if you try to process any file under SCCS that has multiple links.

To prevent simultaneous updates to SCCS files, when an x-file is created a **lock-file**, called the **z-file**, is also created. A z-file contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Other SCCS commands that modify SCCS files will not process an SCCS file if a corresponding z-file exists. For example, assume that two people are editing two versions of an SCCS file. When one of them executes *delta*, a z-file is created which keeps the second person from successfully invoking *delta*. When *delta* has completed, the z-file is removed and the second person is free to create his own *delta*. Z-files are created with mode 444 (read only) in the directory containing the SCCS files and are owned by the effective user.

SCCS checks for the corruption of an SCCS file by maintaining a checksum. Whenever the file is modified with an SCCS command, its checksum is updated to reflect the logical sum of the number of characters the file has. Most SCCS commands will not allow you to access a file that is corrupted. The *admin* command allows you check for corrupted files and to correct them.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files since most of the commands allow you to operate on all of the SCCS files in a directory by merely specifying a directory name. The contents of directories should correspond to convenient logical groupings, such as subsystems of a large project.

System Protection Using Admin

Admin allows the system administrator of a project to control five major areas of protection:

1. Prohibit concurrent editing of one version of a file;
2. Specify a list of users that have permission to edit a file;
3. Prohibit editing on particular releases;
4. Set range limits to what releases users can access;
5. Make the recognition of no ID keywords in a file by SCCS commands a fatal error.

The *admin* command allows you to use these protection strategies on either a file-by-file basis or on a directory basis. How this is done is discussed in a previous section "Using Admin".

Using SCCS With Make

If you are using *make* to create and maintain systems and are using SCCS to maintain the source files for the systems, you can make the two work together by including SCCS commands in *make's* makefiles. The following discussion assumes that you already know how to use *make*. You can refer to its entry in the *HP-UX Reference* or the article on it in *HP-UX Concepts and Tutorials* for information about it.

There are a few basic targets that most makefile should have. These are:

a.out	(or whatever the makefile generates.) This target entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates several intermediate things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special <i>chmod's</i> or <i>ranlib's</i> as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes unneeded files from the directory.

The clean entry should not remove files that can be regenerated from the SCCS files since it is sufficiently important to have the source files around at all times.

Note that the examples of makefiles that follow are only partial and do not illustrate all of these target entries fully. Also note that the example makefiles require that you execute *make* in the same directory as the SCCS files.

To Maintain Groups of Programs

Frequently there are directories with several largely unrelated programs (such as simple commands) and these can often be maintained by one makefile. For example, the makefile below maintains “prog” and “example”:

```
LDFLAGS= -i -s

prog: prog.o
    $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h

example: example.o
    $(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
    set s,$<
```

Note that the source for the programs is maintained as SCCS files and that these files must exist in the same directory as the makefile for the makefile to be able to retrieve them. The .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the .o file on the .c file is important. Another way of doing the same thing is:

```
SRCS= prog.c prog.h example.c

LDFLAGS= -i -s

prog: prog.o
    $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h

example: example.o
    $(CC) $(LDFLAGS) -o example example.o

sources: $(SRCS)
$(SRCS):
    set s,$@
```

There are some advantages to the second approach:

- the explicit dependencies of .o files on .c files are not needed;
- there is an entry called “sources” so if you just want to get all the sources you can just say “make sources”;
- the makefile is less likely to do confusing things since it won’t try to get things that do not exist.

To Maintain a Library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, *make* can adequately handle libraries that are in the process of being developed. One problem in maintaining libraries is that the object (".o") files must be kept out of the library as well as in the library.

```
# configuration information
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.c d.s x.h y.h z.h
TARG=  /usr/lib

# Programs
GET=   get
REL=
AR=    -ar
RANLIB= ranlib

lib.a: $(OBJS)
       $(AR) rvu lib.a $(OBJS)
       $(RANLIB) lib.a

install: lib.a
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) s,$@

Print: sources
        pr *.h *.c[s]

clean:
        rm -f *.o
        rm -f core a.out $(LIB)
```

The "\$\$(REL)" in the \$(SRCS) entry allows you to retrieve various versions of the SCCS files. For example:

```
make REL=-r1.3
```

Note that for the install entry to execute properly, no one should be editing any of the SCCS files when it is invoked.

To Maintain a Large Program

Consider this example makefile:

```
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.y d.s x.h y.h z.h

GET=   get
REL=

a.out: $(OBJS)
       $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
       $(GET) $(REL) s,$@
```

(The print and clean entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules are recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
     touch z.h
```


This would be used in cases where file *z.h* has a line:

```
#include "x.h"
```

in order to bring the date of *z.h*'s last modification in line with the date of the last modification of *x.h* (or rather, when the system thinks *z.h* was last modified). Alternatively, the effect of the *touch* command can be achieved by doing a *get* on *z.h*.


Using SCCS on a Multi-User Project

This section describes how SCCS is configured to maintain files for a large project that involves several users. The person that configures and controls the SCCS files is called the "SCCS System Administrator". You only need the information covered in this section if you are your project's SCCS System Administrator.



If you plan to use SCCS on a project that involves several users, you must first develop a system of controlling access to the SCCS files and commands. Thus far, this tutorial has only discussed a one-user system, where that one user has write access to the directory containing the SCCS files. The user has full use of all of the SCCS commands and can modify protected files (by first making read-only files writeable).


As an SCCS System Administrator, you should provide an interface program that gives temporary write access to the SCCS directory when users execute certain SCCS commands and you should restrict the users to read-only access at all other times. When SCCS files used on a project, they are grouped in one directory (or more if necessary). The SCCS System Administrator is the owner of the SCCS directory, has write access to it, and has full use of all of the SCCS commands. Other users involved on the project should only have read access to the directory, which means that they can not directly use the SCCS commands that require write access.



The SCCS interface program is a C program that provides a filter for the commands requiring that the user have directory write access. If instead of using the interface program you give all of the users write access to the SCCS directory, you greatly restrict the protection facilities SCCS provides. Use of the interface provides users with only temporary write access when they execute one of the commands. The two SCCS commands that require directory write access and that must be available to the users through the interface program are *get* and *delta*. *Rmdel*, *cdc*, and *unget* also require write access and can also be made available to users through the program. The remaining SCCS commands either do not require write access to the SCCS directory or are usually used only by the SCCS System Administrator (for example, *admin*).

How the SCCS Interface Works

The SCCS interface program invokes a specified SCCS command and causes the command's process to inherit the privileges of the SCCS System Administrator for the duration of its execution. This allows the process to obtain write access to the SCCS directory.



The names of the commands that you want filtered through the interface program must be linked to the program so that invoking the command name executes the program. The interface program is written in C and when a C program is executed, the name that invoked the program is passed as argument 0 and is followed by any user-supplied arguments. By looking at the value of argument 0, the program knows which command to execute. Thus, the command name used to invoke the interface program determines which SCCS command the program executes. How other arguments, such as SCCS file names, are processed is often system dependent, but they can be passed directly to the SCCS command by the program.

Configuring an SCCS System Using the Interface

As the SCCS System Administrator, there are six basic steps that you must carry out before allowing other users to access SCCS files:

1. Create and move to an SCCS directory.
2. Write and compile the interface program.
3. Change the mode of the program.
4. Set up links between the program and the SCCS command names.
5. Modify each user's search path so that the directory containing the interface program is searched before "/usr/bin", the directory containing the SCCS commands.
6. Create the SCCS files.

Creating the SCCS Directory

Before you can successfully use the SCCS interface program, you must create one or more directories for storing the SCCS files and the program. You, as the SCCS System Administrator, should be the only one with write access to the directory.

For example, to create a directory called "/system/sccs" and then restrict write access to yourself, use:

```
mkdir /system/sccs
chmod 755 /system/sccs
```

You must now move to the SCCS directory since you must to write and maintain the SCCS interface program there:

```
cd /system/sccs
```

Writing and Compiling the Program

The SCCS interface program is written in C and this section assumes that you already know how to program in that language.

You should write an SCCS interface program that is customized to the needs of your system. To get you started, however, a general purpose interface program is provided below.

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i; /*counts command line arguments*/
    character cmdstr[LENGTH]; /*holds SCCS command name*/

    /*
     * Do any required processing of file name arguments that
     * follow the SCCS command name (arguments that don't begin
     * with -)
     */

    for (i = 1; i < argc; i++)
        if (argv[i][0] != '-')
            argv[i] = filearg(argv[i]);
}
```

```

/*
   Get "simple name" of name used to invoke this Program
   (i.e. strip off directory-prefix name, if any).
   This step may not be needed in your system.
*/

argv[0] = sname(argv[0]);

/*
   Invoke actual SCCS command, passing arguments.
*/

sprintf(cmdstr, "/usr/bin/%s", argv[0]);

execv( cmdstr, argv);
}

```

This example program calls two routines that you must supply and that allow you to customize the SCCS interface. "Filearg" acts as a preprocessor for SCCS commands. In the program above, it is used to modify SCCS file name. This modification often involves appending the path name of an SCCS directory to the SCCS file names so that users can access the files without having to specify full path names.

The second routine that you must supply is "sname". Its purpose is to modify the name with which the user invoked the interface program so that it agrees with the name of the associated SCCS command. The statement calling this routine is not required when the link names of the interface program are the same as the names of the SCCS commands.

Once you have written an SCCS interface program designed for your system, you must compile it. Assuming that you source code file is called "interface.c", use the following to compile it:

```
cc interface.c -o interface
```

The name of the resulting executable program is "interface".

Specifying the Mode of the Program

The interface program must be owned by the SCCS System Administrator, and must be executable by the other users involved on the project. It must also have its "set user ID on execution" bit on so that when the program is executed, the user obtains write access to the SCCS directory. Assign these necessary characteristics to the program with:

```
chmod 4755 interface
```

where "interface" is the name of the executable interface program.

Assign Name Links to the Program

Now that you have an executable interface program, use the `cp` command to assign name links to it. It is convenient for the users if these name links are the same as the SCCS commands that are executed by the program.

To illustrate, assume that you want to allow users to access the *get* and *delta* commands through the interface program. Create the necessary links with:

```
cp interface get
cp interface delta
```

You now have three names that point to the same program: “interface”, “get”, and “delta”. All of the other SCCS commands that require write access to the SCCS directory will be inaccessible to the users since you have not linked them to the program.

Modifying the Users' Search Path

Once you have linked the appropriate SCCS command names to the SCCS interface program, you must modify each user's HP-UX search path so that the directory containing the the interface program is found before the actual SCCS commands. PATH is the HP-UX variable that specifies where the system looks for a command when a user executes it. When any command is executed, the system searches for the command in the directories defined by the user's PATH variable. The directories are searched in the order in which they appear in the variable's list. Your HP-UX system has a default definition for PATH but it can be redefined by each user in his “.profile” file. Refer to your system's *HP-UX System Administrator Manual* for more information about the PATH variable and the “.profile” file.

Whether you have to change the PATH variable in every user's “.profile” file or just the system's default definition, you must insert the SCCS directory name before the appearance of “/usr/bin”, the directory containing the SCCS commands, in PATH's directory list. For example, if a user's PATH variable is defined as:

```
PATH=/bin:/usr/bin
```

you should change it to:

```
PATH=/bin:/system/sccs:/usr/bin
```

where “/system/sccs” is the name of the SCCS directory containing the SCCS interface program. When you execute a command, the system first searches for it in /bin, then in /system/sccs, and finally in /usr/bin.

Creating SCCS Files

As SCCS System Administrator, you are the only user able to execute *admin* because it requires write access to the SCCS directory and you did not specify it as a link name to the SCCS interface program. Having sole access to *admin* means that you can strictly control the creation of SCCS files and the setting to their various flags. Refer back to the section “SCCS's Protection Facilities” in this tutorial for more information.

Note that in order to make full use of SCCS for a multi-user project, SCCS files should be maintained in a central location and logically grouped into one or more SCCS directories.

Quick Reference

Commands

In the discussion of the following SCCS commands, only the most useful keyletter arguments are discussed. Refer to the *HP-UX Reference* for complete descriptions of the commands and all of their keyletters.

- get* Gets files for compilation (not for editing). ID keywords are expanded. Note that *get -e* is listed separately.
- rSID Version to get.
 - p Send text to standard output rather than to the actual file.
 - k Don't expand ID keywords.
 - ilist List of deltas to include.
 - xlist List of deltas to exclude.
 - m Precede each line with SID of creating delta.
 - cdate Don't apply any deltas created after date.
- get -e* Gets files for editing. ID keywords are not expanded. Should be matched with a *delta* command.
- rSID Same as *get -rSID*. If SID specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with SID.
 - b Create a branch.
 - ilist Same as *get -ilist*.
 - xlist Same as *get -xlist*.
- delta* Merge a file retrieved with *get -e* back into the s-file. Collect comments about why this delta was made.
- unget* Remove a file previously retrieved with *get -e* without merging the changes into the s-file.
- prs* Print information about the SCCS file.
- sact* Determine who is currently editing a file.
- what* Find and print ID keywords that have been expanded. They must be preceded by *@(#)* (the expand form of the keyword %Z%).

admin

Create or set parameters on s-files.

- ifile Create s-file, using file as the initial contents.
- z Rebuild the checksum in case the file has been trashed.
- fflag[value] Turn on the flag and optionally give it a value.
- dflag Turn off (delete) the flag.
- tfile Replace the descriptive text in the s-file with the contents of file. If file is omitted, the descriptive text is deleted from the s-file. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.
- h Check for corruption in the s-file.

Useful flags are:

- b Allow branches to be made using the -b flag to get -e.
- dSID Default SID to be used on a get.
- i Cause "No Id Keywords" error message to be a fatal error rather than a warning.
- t The module "type"; the value of this flag replaces the %Y% keyword.

sccsdiff

Compare two versions of an SCCS file.

cdc

Change the comment line or MR number associated with a previously created delta.

rmdel

Remove a delta from an SCCS file. This delta must be the most recent on its branch or the main trunk-- no other deltas can depend on it.

help

Supplies additional information about an SCCS error message.

ID Keywords

%Z%	Expands to “@(#)” for the <i>what</i> command to find. Every ID keyword string that you want <i>what</i> to see must be preceded by this keyword.
%M%	The current module name, e.g., “prog.c”. Unless set by <i>admin</i> , it defaults to the file name minus the “s.” prefix.
%F%	The SCCS file name.
%Y%	The value of the <i>t</i> flag as set by <i>admin</i> .
%I%	The SID of the retrieved text. The highest delta applied.
%W%	A shorthand for “%Z% <i>M</i> % <tab> %I%”.
%E%	The date of the delta corresponding to the “%I%” keyword (YY/MM/DD).
%G%	The date of the delta corresponding to the “%I%” keyword (MM/DD/YY).
%U%	The time the delta corresponding to the “%I%” keyword was created (HH:MM:SS).
%R%	The current release number, i.e., the first component of the “%I%” keyword.
%L%	The current level number, i.e., the second component of the “%I%” keyword.
%B%	The current branch number, i.e., the third component of the “%I%” keyword, if it exists.
%S%	The current sequence number, i.e., the fourth component of the “%I%” keyword, if it exists.
%D%	The current date (YY/MM/DD).
%H%	The current date (MM/DD/YY).
%T%	The current time (HH:MM:SS).
%Q%	The value of the <i>q</i> flag as set by <i>admin</i> .
%C%	The current line number. It is intended for identifying messages output by the program such as “this shouldn’t have happened” type errors. It is <i>not</i> intended to be used on every line to provide sequence numbers.

Table of Contents

Chapter 1: Interfacing Concepts

Introduction	1
Manual Organization	1
The DIL Interfacing Routines	2
Location of the Interfacing Routines	2
Linking the DIL Routines	2
Calling the DIL Routines From Pascal	2
Calling the DIL Routines From FORTRAN	3
Why Do You Need an Interface?	4
Electrical and Mechanical Compatibility	5
Data Compatibility	5
Timing Compatibility	5
Additional Interface Functions	5
The HP-IB Interface	6
General Structure	6
Handshake Lines	6
Bus Management Control Lines	7
The GPIO Interface	8
Data Lines	8
Handshake Lines	8
Special Purpose Lines	8
Data Handshake Methods	9
Data-In Clock Source	9

Chapter 2: General Purpose Routines

Computer Communication With an Interface	12
Creating an Interface File	12
Opening the Interface Files	15
Closing the Interface File	16
Reading and Writing	17
Designing Error Checking Routines	18
Using Errno	18
Resetting Interfaces	19
Controlling I/O Parameters	20
Setting the I/O Timeout	20
Setting Data Path Width	21
Setting Transfer Speed	22
Setting Read Termination Character	23
Removing a Read Termination Character	24
Determining Why a Read Terminated	25
Interrupts	27

Chapter 3: Controlling the HP-IB Interface

Overview of HP-IB Commands	31
Overview of the HP-IB DIL Routines	35
The Computer's Role on the HP-IB	36

Opening the HP-IB Interface File	37
Sending HP-IB Commands	38
The Active Controller	40
Determining Active Controller	40
Setting Up Talkers and Listeners	41
Remote Control of Devices	43
Locking Out Local Control	43
Enabling Local Control	43
Triggering Devices	44
Transferring Data	44
Clearing HP-IB Devices	45
Servicing Requests	46
Parallel Polling	47
Waiting For a Parallel Poll Response	51
Serial Polling	54
Passing Control	55
The System Controller	56
Determining System Controller	56
System Controller's Duties	57
The Computer As a Non-Active Controller	59
Determining the Controller's Status	59
Requesting Service	60
Responding to Parallel Polls	61
Disabling Parallel Poll Response	63
Accepting Active Control	63
Determining When You Are Addressed	65
Buffering I/O Operations	68
Iodetail: The I/O Operation Template	68
Allocating Space	70
An Example	71
Locating Errors in Buffered I/O Operations	74

Chapter 4: Controlling the GPIO Interface

Configuring Your GPIO Interface	75
Setting the Interface Switches	75
Creating the GPIO Interface File	75
Limitations on Controlling the Interface	76
Using the DIL Routines	76
Resetting the Interface	77
Performing Data Transfers	77
Using the Special-Purpose Lines	78
Controlling the Data Path Width	79
Controlling the Transfer Speed	79
Read Terminations	80
Interrupts	80

Appendix A: Series 500 Dependencies

Creating the Interface File	81
Determining the Driver	81
Determining the Select Code	81

Determining The Bus Address of the Interface Card	82
Entity Identifiers	82
Restrictions Using the DIL Routines	82
hpib_bus_status	82
hpib_card_ppoll_resp	83
hpib_rqst_srvce	83
hpib_send_cmnd	83
hpib_status_wait	84
hpib_wait_on_ppoll	84
io_get_term_reason	84
io_timeout_ctl	85
io_speed_ctl	85
io_width_ctl	85
Performance Tips	86
Appendix B: Character Codes	89

Index



Interfacing Concepts

Introduction

This tutorial illustrates how to access an arbitrary device through **HP-IB** (Hewlett-Packard Interface Bus) and **GPIO** (General Purpose Input/Output) interfaces on your HP-UX system using the routines in **DIL** (Device I/O Library). This tutorial covers general interfacing strategies, in addition to strategies designed specifically for HP-IB and GPIO interfaces.

The tutorial assumes that you want to communicate with devices from within a program process. All DIL routines can be called from C, Pascal, and FORTRAN programs. The examples illustrating the use of the routines are written in C; however, with a little extra code they can be accessed from Pascal or FORTRAN programs.

Manual Organization

Chapter 1: Interfacing Concepts presents basic interfacing concepts and a description of the HP-IB and GPIO interfaces.

Chapter 2: General Routines discusses how the interfaces are accessed in the HP-UX environment and how basic data transfers are implemented.

Chapter 3: Controlling the HP-IB Interface describes interfacing techniques for the HP-IB interface.

Chapter 4: Controlling the GPIO Interface covers interfacing techniques for the GPIO interface.

Appendix A: Series 500 Dependencies covers hardware and system dependent information. Check for restrictions on using the DIL routines by referring to the appendix for your system.

Appendix B: Character Codes

The DIL Interfacing Routines

Location of the Interfacing Routines

The DIL routines that provide direct control of your computer's interfaces are contained in the `/usr/lib/libdvio.a` library. Some routines are general purpose and can be used with any interface supported by the library, while others provide control of specific interfaces. The Device I/O Library (DIL) currently supports the HP-IB and GPIO interfaces.

Linking the DIL Routines

You can make calls to the DIL routines from C, Pascal, or FORTRAN programs. However, the library is not automatically linked with your program when you compile the program with `cc`, `pc`, or `fc`. You must use the `-l` flag to specify that the library be linked with the program. To compile a C program and then link the DIL routines with it, type:

```
cc program.c -ldvio
```

To do the same thing with a Pascal program, type:

```
pc program.p -ldvio
```

and with a FORTRAN program, type:

```
fc program.f -ldvio
```

In all three cases, the `-l` option is passed to the HP-UX linker, causing it to look for and search a library named `/lib/libxxx.a` where `xxx` is a string of characters specified after the `-l` option. If the linker fails to find the routine in that library it continues to search for it in `/usr/lib/libxxx.a`. If you do not indicate the `/usr/lib/libdvio.a` be linked with your program, any calls to DIL routines are seen as errors by the HP-UX linker.

Calling the DIL Routines From Pascal

External declarations are required for each DIL routine that you want to call from a Pascal program. This declaration consists of the routine heading, including a formal parameter list and result type, followed by the pre-defined word `EXTERNAL`. For example, the C description of `open` is:

```
int open( path, oflag);
char *path;
int oflag;
```

The external declaration in a Pascal program for the routine looks like:

```
TYPE
  PATHNAME = PACKED ARRAY [0..50] OF CHAR;

FUNCTION open
  (VAR path: PATHNAME;
   oflag: INTEGER):
  INTEGER;
  EXTERNAL;
```

Note that one of the parameters in the *open* declaration is type *VAR*, indicating that the value be passed by reference. This simulates the passing of a pointer, which is what *open* expects. There is generally a straight forward mapping between a routines C declaration and Pascal equivalent.

Calling the DIL Routines From FORTRAN

C and FORTRAN routine calls are not compatible because C passes parameters **by value** while FORTRAN passes them **by reference**.

To overcome this incompatibility, you direct the compiler to generate a call by value using FORTRAN's *\$ALIAS* directive. For example:

```
$ALIAS close = 'close' (%val)
```

If your system's FORTRAN compiler does not support this form of *\$ALIAS*, you may need to solve the parameter passing differences by writing an **onionskin** routine in either C or Pascal. An **onionskin** is a C language function written for the purpose of resolving parameter passing irregularities between C and other languages.

For example, to access *close* using an onionskin routine you use:

```
$ALIAS close = '_my_io_close'
```

and then write the onionskin routine:

```
int my_io_close (eid);  
/*the compiler will create the external symbol "_my_io_close"  
based on the above declaration*/  
int *eid;  
{  
    int real_eid = *eid;  
    return (close (real_eid));  
}
```

Why Do You Need an Interface?

The remainder of this chapter presents a brief description of what an interface is and focusses on the HP-IB and GPIO interfaces in particular. This information is provided for background purposes only, it is not required before using the DIL routines. You can skip the remainder of this chapter and go immediately to *Chapter 2: General Purpose Routines*.

The primary function of an interface is providing a communication path for data and commands between the computer and its resources. Interfaces act as intermediaries between resources by handling part of the **bookkeeping** work and ensuring that this communication process flows smoothly. The following paragraphs explain the need for interfaces.

First, even though the computer backplane is driven by electronic hardware that generates and receives electrical signals, the hardware was not designed to be connected directly to external devices. The electronic backplane hardware was designed with specific electrical logic levels and drive capability in mind. Exceeding its ratings damages this electronic hardware.

Second, in assuming that connectors and signals are compatible, you have no guarantee data sent is interpreted properly by the receiving device. Some peripherals expect single-bit serial data while others expect 8-bit parallel form.

Third, there is no reason to believe that the computer and peripheral are in agreement as to when data transfer occurs; and when the transfer does begin, the transfer rates probably will not match. As you can see, interfaces are responsible for overseeing the communication between computer and its resources. The functions of an interface are shown in the following block diagram (see Figure 1.1).

Fourth, you cannot be assured that the connectors of the computer and peripheral are compatible. In fact, there is good probability the connectors may not mate properly, let alone provide a one-to-one correspondence between the function of each signal wire.

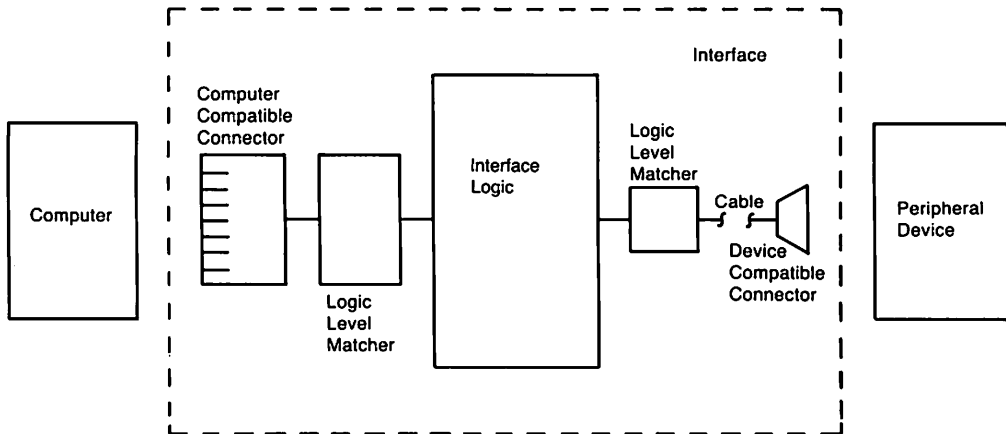


Figure 1.1: Functional Diagram of an Interface

Electrical and Mechanical Compatibility

Electrical compatibility must be ensured before connecting two devices. Often two devices have input and output signals that do not match. If so, the interface strives to **match** the electrical **logic levels** of these signals before the physical connections are made.

Mechanical compatibility simply means that the connector plugs must fit together properly. All HP computer interfaces have connectors that mate with the computer backplane. The peripheral end of the interfaces can have unique configurations due to the fact that several types of peripherals are available. Cables are available for most interfaces that will connect directly to the device so you don't have to wire the connector yourself.

Data Compatibility

Just as two people must speak a common language, the computer and peripheral must agree upon the form and meaning of data before communicating. As a programmer, one of the most difficult compatibility requirements to fulfill (before exchanging data) is making sure the format and meaning of the data being sent is identical to that anticipated by the receiving device. Even though some interfaces format data, most interfaces are not responsible for matching data formats. Most interfaces merely move agreed upon quantities of data to or from computer memory. The computer must generally make the necessary changes, if any, so that the receiving device gets meaningful information.

Timing Compatibility

Since all devices do not have standard data-transfer rates, nor agree to when the transfer will take place, a consensus between sending and receiving devices must be made. If the sender and receiver agree on both the transfer rate and beginning point (in time), the transfer can be made easily.

The data transfer must be started at an agreed-upon time and at a known rate. If not, the transfer proceeds one data item at a time with the receiving device acknowledging that it received the data and that the sender can transfer the next data time. This process is known as a **handshake**. Both types of transfers are utilized with different interfaces.

Additional Interface Functions

Another powerful feature of an interface card is relieving the computer of low-level tasks, such as performing data transfer handshakes. This distribution of tasks eases some of the computer's burden and decreases the otherwise stringent response-time requirements of external devices. The actual tasks performed by each type of interface card varies widely. The next sections concentrate on the functions of two particular interfaces: the HP-IB and the GPIO.

The HP-IB Interface

The **HP-IB** (Hewlett-Packard Interface Bus) provides compatibility between the computer and external devices conforming to the IEEE 488-1978 standard. Electrical, mechanical, and timing compatibility requirements are satisfied by the bus which allows you to connect up to 15 devices to one interface.

General Structure

Communications through the HP-IB are made according to a precise set of rules defined by the IEEE 488-1978 standard. These rules ensure orderly communication.

There are three types of devices on the HP-IB:

- controller
- talker
- listener

These types are actually attributes that exist alone or in combinations in one device. For example, the HP-IB interface allows a desktop computer to be a **controller**, **talker**, and **listener**. A device that accepts data from the bus (for example a printer) is usually a **listener**, while a device that supplies data to the bus (for example a voltmeter) is usually a **talker**. At any one time, the bus has only one Active Controller and only one talker, but it can have any number of listeners.

The HP-IB is composed of 16 lines which are divided into 3 groups. 8 lines form a bi-directional data path which carries data, commands, and device addresses; 3 lines control the transfer of data bytes; and the 5 remaining lines control bus management.

Handshake Lines

The **handshake** lines used to synchronize data transfers are:

- DAV - Data valid
- NRFD - Not ready for data
- NDAC - Not data accepted

The HP-IB interlocking handshake uses the lines as follows. All devices currently designated as active listeners indicate when they are ready for data using the **NRFD** line. A device that is not ready to receive data **asserts** this line (by pulling it low). Any device that is ready lets the line **float** high. Since an active low overrides a passive high, the NRFD line stays low until all active listeners are ready for data.

When the bus talker senses all devices are ready, it places the next data byte on the data lines and asserts **DAV** by pulling it low. This tells the listeners that the information on the data lines is **valid** and they can read it. Each listener then accepts the data and lets the **NDAC** line float high (false). As with NRFD, when all listeners have let NDAC go high the talker **senses** that all listeners have read the data. It can then float DAV (let it go high) and start the entire sequence over again for the next byte of data.

Bus Management Control Lines

There are five bus management control lines:

- ATN – Attention
- IFC – Interface clear
- REN – Remote enable
- EOI – End or identify
- SRQ – Service request

ATN: The Attention Line

Command messages are encoded on the data lines as 7-bit ASCII characters, and are distinguished from the normal data characters by the attention line's (**ATN**'s) logic state. That is, when **ATN** is false, the states of the data lines are interpreted as data. When **ATN** is true, the data lines are interpreted as commands.

IFC: The Interface Clear Line

Only the System Controller sets the **IFC** line true. By asserting **IFC**, all bus activity is unconditionally terminated, the System Controller becomes the Active Controller, and any current talker and listeners become unaddressed. Normally, this line is used to terminate all current operations, or to allow the System Controller to regain control of the bus. It overrides any other activity currently taking place on the bus.

REN: The Remote Enable Line

This line allows instruments on the bus to be programmed remotely by the Active Controller. Any device addressed to listen while **REN** is true is placed in its remote mode of operation.

EOI: The End or Identify Line

The **EOI** line is used to indicate the end of a data message. Normally, data messages sent over the HP-IB are sent using standard ASCII code and are terminated by the ASCII line-feed character. However, certain devices need to send blocks of information containing data bytes which have the line-feed character bit pattern as part of the data message. Thus, no bit pattern can be designated as a **terminating character**, since it could occur anywhere in the data stream. For this reason, the **EOI** line is used to mark the end of the data message.

Another function of **EOI** is that, when it is asserted along with the **ATN** line, a parallel poll is taken of the bus.

SRQ: The Service Request Line

The Active Controller is always in charge of ordering events that occur on the HP-IB. If a device on the bus needs the Active Controller's help, it sets the **SRQ** line true. The **SRQ** line sends a request for service, not a demand, and it is up to the Active Controller to choose when and how it services the request. However, the device continues to assert **SRQ** until it has been satisfied. Exactly what satisfies a service request depends on the requesting device, and is explained in the device's operating manual.

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. On the Series 500, the interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user definable signal lines provide additional flexibility.

The GPIO interface is composed of the following lines:

- 16 parallel data input lines
- 16 parallel data output lines
- 4 handshake lines
- 4 special purpose lines

Data Lines

There are 32 data lines: 16 for input and 16 for output. These lines normally use negative logic (0 indicates true, 1 indicates false). The logic can be changed so that a 1 indicates true with the interface's Option Switches. Refer to your GPIO interface manual to see how to do this.

Handshake Lines

Although four lines fall into this group, only three are used for controlling the transfer of data:

- **PCTL** – Peripheral Control line
- **PFLG** – Peripheral Flag line
- **I/O** – Input/Output line

The Peripheral Control (**PCTL**) line is controlled by the interface and used to initiate data transfers. The Peripheral Flag (**PFLG**) line is controlled by the peripheral device and used to signal the peripheral's readiness to continue the transfer process. The Input/Output (**I/O**) line is used to indicate direction of data flow.


The fourth handshake line is the External Interrupt Request (**EIR**) line. This line is used by a peripheral to signal service requests to the computer.

Special Purpose Lines

Four lines are available for any purpose you desire; two are controlled by the peripheral device and sensed by the computer, and two are controlled by the computer and sensed by the peripheral.

Data Handshake Methods


Handshaking is a method of synchronizing transfer of data from the sending device to the receiving device. In order to use any handshake method, the computer and peripheral device must be in agreement as to how and when several events occur.



There are two handshake methods using PCTL and PFLG to synchronize data transfers: **Pulse-Mode Handshakes** and **Full-Mode**. If the peripheral uses pulses to handshake data transfers and meets certain hardware timing requirements, the Pulse-Mode Handshake is used. The Full-Mode Handshake should be used if the peripheral does not meet the Pulse-Mode timing requirements. Refer to the GPIO interface's documentation for a description of these handshake methods.

Data-In Clock Source

Ensuring that data is **valid** when read by the receiving device differs slightly depending on what direction the data is flowing. When **writing data out** from the computer the interface generally holds data valid while PCTL is in the asserted state, the peripheral must read the data during this period. When **reading data from** the peripheral, the peripheral must hold the data valid until it can signal that the data is valid or until the data is read by the computer. The peripheral signals that the data is valid using the PFLG line. This clocks the data into the interface's Data-In registers.



You can specify the logic level of the PFLG line that indicates valid data by setting the **FLAG** switches on the interface card. Refer to the card's installation manual to find out how to do this.



General Purpose Routines

2

The Device I/O Library (DIL) contains eight routines that can be used with any interface supported by the library. The table below lists them.

Routine	Description
<i>io_reset</i>	Reset the interface associated with the interface file having the specified entity identifier.
<i>io_timeout_ctl</i>	Assign a timeout value for data transfers.
<i>io_width_ctl</i>	Set the width of the data path for the interface associated with the interface file having the specified entity identifier.
<i>io_speed_ctl</i>	Select a transfer speed for the interface associated with the file having the specified entity identifier.
<i>io_eol_ctl</i>	Set up a read termination character on an interface file associated with the specified entity identifier.
<i>io_get_term_reason</i>	Determine how the last read terminated for the file associated with the specified entity identifier.
<i>io_on_interrupt</i>	Set up interrupt handling for program.
<i>io_interrupt_ctl</i>	Allow enabling and disabling of interrupts for the associated <i>eid</i> .

This chapter presents some techniques for using these routines in I/O processes, along with four system routines: *open*, *close*, *read*, and *write*. However, before you can use them, you must create a file HP-UX can use to communicate with the interface.

Computer Communication With an Interface

HP-UX treats I/O to an interface in the same way it treats I/O to a file. In fact, before your computer can communicate with an interface, a **special file** must be created. Normally, this special file defines the location of a particular device connected to the interface and the manner in which the computer and the device communicate. However, to use the routines in DIL, you must set up a special file for the interface, not for a device on the interface.

The general process of creating special files is described in the *HP-UX System Administrator Manual* for your system. The following discussion points out specific requirements needed for a special file associated with an interface. Refer to the Appendix for hardware-dependent information.

Creating a special file for an interface is the one step in the interfacing process described in this manual that is not done from within a program process. Special files are created from either your HP-UX shell or a shell script.

Creating an Interface File

Special files are created with *mknod*. To use this command, first log onto your system as the super-user. Next, determine the values of the following parameters required by *mknod*:

1. Determine a file **pathname** that identifies the interface itself, not a peripheral on the interface. Special files are usually kept in the directory */dev*. This is basically a HP-UX convention; some commands expect to find special files in the */dev* directory and fail if they are not there.
2. Determine which driver must be used to talk to the interface. The driver is specified with a **major** number. This information can be found in the Appendix for your system and in the *HP-UX System Administrator Manual*.
3. On the Series 500, determine a **minor** number, a hexadecimal value specifying the location of the interface. It has the following form:

`0xScAdUV`

where:

- `0x` specifies that the characters which follow represent hexadecimal values. These two characters (zero and x) are entered as shown.
- `Sc` a two-digit hexadecimal value specifying the select code of the interface card.
- `Ad` a two-digit hexadecimal value specifying a bus address. If the interface normally handles automatic addressing to a particular device (e.g. the HP-IB interface), the value you specify here places the interface in **raw** mode so it does no automatic addressing. Refer to the Appendix for your system to find out what value is required. If only one device can be connected to the interface (e.g. the GPIO interface), the component of the minor number is ignored.
- `U` a single-digit hexadecimal value specifying a secondary address. The component of the minor number is ignored when the special file you are creating is for an interface.
- `V` a single-digit hexadecimal value specifying a secondary address, such as the volume number in a multi-volume drive, which is ignored also.

Once you have values for the pathname, major number, and minor number, you can execute *mknod* from your HP-UX shell. The command line has the form:

```
mknod pathname c major minor
```

Note that *c* indicates you want to create a **character special file**. This is required if you plan to access the interface using DIL routines.

Creating an HP-IB Interface File

Normally, HP-IB device files refer to a specific device or bus. Since the DIL routines are restricted to working with the interface card, a special device file that refers to the interface must be used. This device file will be referred to as a **raw** HP-IB device file. **Raw** refers to bytes of data that are uninterpreted, in this case, by the interface.

Assume that you have an HP-IB interface that you want to access using DIL routines on a Series 500 computer. To do this you must first create a character special file for it using *mknod*.

1. Log onto your Series 500 as the super-user.
2. Choose a pathname for the interface. For this example, use */dev/raw_hpib*.
3. You must create a character special file so that the DIL routines can be used; therefore, the file type argument of *mknod* is *c*.
4. Referring to *Appendix A: Series 500 Dependencies* you find that the raw HP-IB interface must use driver 12 which is built-in. The Series 550 uses driver 37 for the HP-IB interface. Therefore, the major number argument of *mknod* is 12 or 37.
5. Determine the minor number argument for *mknod*. One component of the minor number depends on the interface's **select code**. On the Series 500 this code corresponds to the I/O slot of the interface card. Assume that you have placed the card in the slot with select code 02. Another component specifies a **bus address**. To place the HP-IB interface in raw mode you must use an address of 31. Since the minor number requires hexadecimal values, this component is *1f*. The last two components of the minor number specifying secondary addresses are ignored. (They are set to zero in the call to *mknod* below.)
6. Now invoke *mknod* with:

```
mknod /dev/raw_hpib c 12 0x021f00
```

You should now log out as super-user.

If you study the previous procedure, you notice that for all raw HP-IB interface files you create on a particular computer only two values vary: the pathname of the interface file and the select code component of the minor number. To illustrate, assume that you have two HP-IB interface cards installed for Series 500 and you want to access them both using DIL routines. The commands below set up a special file */dev/raw_hpib1* at select code 02 and a special file */dev/raw_hpib2* at select code 03:

```
mknod /dev/raw_hpib1 c 12 0x021f00
```

```
mknod /dev/raw_hpib2 c 12 0x031f00
```

Creating a GPIO Interface File

Now assume that you have a GPIO interface that you want to access with the DIL routines on the same Series 500 computer. Using the following steps, you can create a character special file for the interface.

1. Log onto your Series 500 as the super-user.
2. Choose a pathname for the interface. For this example, use `/dev/raw_gpio`.
3. You must create a character special file so that the DIL routines can be used; therefore, the file type argument of `mknod` is `c`.
4. Referring to *Appendix A: Series 500 Dependencies* you find that the GPIO interface must use driver 18; therefore, the major number argument of `mknod` is 18.
5. Because the GPIO interface is not a bus architecture, the usual bus address and secondary address components of `mknod`'s minor number are ignored and you need only determine the select code value. Assume that you have placed the interface in the I/O slot on your Series 500 corresponding to select code 04.
6. Now invoke `mknod` with:

```
mknod /dev/raw_gpio c 18 0x040000
```

and then log out as super-user. Although the minor number above has its addressing components set to zeros, any hexadecimal values could have been used.

As with the HP-IB interface, only two values vary for each GPIO interface file you create: the pathname of the file and the select code component of the minor number.

Note that the last two examples are for the Series 500 computer.

Opening the Interface Files

Other than the default standard input, standard output, and standard error files, you must explicitly open files in order to read and write to them from inside C, FORTRAN, or Pascal programs. The HP-UX system routine for opening files is *open*. It is called as follows:

```
int  eid;
...
...
eid = open( filename, oflag);
```

filename is either a character string representing a file's external HP-UX name or a pointer to a buffer that contains the external name. The integer *oflag* specifies the access mode for opening the file. It can have one of three possible values: 0 requests read-only access; 1 requests write-only access; and 2 requests both read and write access.

The command *open* returns a non-negative integer (*eid*) that is the entity identifier for the file that you are opening. The entity identifier is an internal name for the file. During any reads or writes to a file you must specify the file by its entity identifier, not its HP-UX file name.

The following code defines an entity identifier called *eid* and opens an interface file called */dev/raw_hpib* with read and write access:

```
int  eid;
eid = open("/dev/raw_hpib", O_RDWR);
```

As an alternative to specifying the character string name of the HP-UX file in the call to *open*, you can place the name in a buffer and then call *open* with a pointer to the buffer. *O_RDWR* (defined in the *include* file *fcntl.h*) is an integer (for example, 2) specifying the access mode for opening the file. For example, the following code also opens the HP-IB interface file:

```
int  eid;
char *buffer;

buffer = "/dev/raw_hpib";
eid = open(buffer, O_RDWR);
```

If a file is successfully opened, *open* returns a non-negative integer as the entity identifier. However, if an error occurs and the file is not opened, a *-1* is returned.

Closing the Interface File

Just as **opening** a file gives a program access to that file, **closing** a file removes access and disconnects the file from the program. Normally, you do not have to worry about closing the files that your program opens because when a program terminates (via *exit* or a return from the main routine) any open files associated with it are closed automatically. However, HP-UX limits the number of files one process (or program) can have open at one time to *NO_FILE*. *NO_FILE* is set to the number of open files allowed and is defined in the *include* file *param.h*. As good programming practice, you should use the HP-UX system routine *close* when you are through using the interface file in a program.

The command *close* requires the entity identifier for an open interface file as an argument. The code below shows how an HP-IB interface can be opened and closed:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open( "/dev/raw_hpib", O_RDWR);
    ...
    ...      /*Can now communicate with the interface*/
    ...
    close(eid);
}
```

The connection between the entity identifier and the open file is now broken and the entity identifier is available for the system to assign to another file. A file that is opened on two separate occasions need not be assigned the same entity identifier both times by the system.

If the routine successfully closes the specified file, it returns a 0; if not, it returns a -1 and *errno* is set to indicate the error (see the section *Designing Error Checking Routines*). A common cause of failure is using an argument in the routine call that is not a valid entity identifier for an open interface file.

Reading and Writing

The lowest level of I/O in HP-UX provides no buffering or other services; it is a direct entry into the operating system. Two HP-UX system routines are available that provide low-level I/O: *read* and *write*. Both require three arguments:

- an entity identifier of an open file
- a buffer in your program where the data is to come from during *write* or go to during *read* (*write* empties a buffer; *read* fills a buffer)
- the number of bytes to be transferred

The call to *read* has this form:

```
#include <fcntl.h>
main()
{
    int eid;          /*the entity identifier*/
    char buffer[10]; /*buffer in which the read data will be placed*/
    eid = open("/dev/raw_hpib", O_RDWR);
    read(eid, buffer, 10); /*reads 10 bytes from a previously opened*/
}                          /*file with the entity identifier "eid". */
```

The call to *write* is very similar:

```
#include <fcntl.h>
main()
{
    int eid;          /*the entity identifier*/
    char buffer[10]; /* the buffer containing data to be written to a file*/
    eid = open("/dev/raw_hpib", O_RDWR);
    buffer = "data message"; /*message to be sent*/
    write(eid, buffer, 10); /*10 bytes are written to previously*/
}                          /*opened file with the entity identifier "eid"*/
```

Although *read* and *write* required the number of bytes to be transferred as their third argument, other parameters, discussed later, associated with the interface file's *eid* can end the transfer before the number is reached.

An Example

Assume that you have already created a special file, */dev/raw_hpib*, for an HP-IB interface. Your program must first open the interface file */dev/raw_hpib* for reading and writing:

```
int eid;
eid = open("/dev/raw_hpib", O_RDWR);
```

To place data on the bus you use *write*:

```
write(eid, "This is a test", 14);
```

The number of bytes to be sent is 14 because there are 14 characters in the data string. To receive 10 bytes of data from the bus you use:

```
char buffer[10];
read(eid, buffer, 10);
```

After *read* has completed, *buffer* contains the 10-byte message.

Designing Error Checking Routines

All Device I/O Library routines return a `-1` to indicate that an error occurred during the routine's execution. If this happens, the routine sets an external HP-UX variable called *errno*.

Errno is an integer variable whose value indicates what error caused the failure of a system or library routine call. It is not reset after successful routine calls; therefore, you should only check its value after you have determined an error occurred. Except for this section, most examples in this manual do not involve checking for the successful completion of routine calls. However, as good programming practice you should include error checking in your own programs.

To make sure that a particular routine sets *errno* if it fails, you should refer to the routine's entry in the *HP-UX Reference*.

Using Errno

To access *errno* from your program you must include the following code at the beginning of the program:

```
#include <errno.h>
extern int errno;
```

The file *errno.h* contains a complete list of error number values for *errno* and their associated names. Refer to the *errno(2)* entry in the *HP-UX Reference* to see this list and to find out the meaning associated with each value.

Once you have included the two lines of code shown above, there are two ways you can check its value if a routine fails. The simplest way is to check to see if the routine failed, and if so, to print out the value of *errno* and then exit. The example below illustrates this strategy:

```
#include <errno.h>
#include <fcntl.h>
extern int errno;
main()
{
    int eid;
    if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1)
    {
        printf("Error occurred. Errno = %d", errno);
        exit(1);
    }
    ...
}
```

If an error occurs and *errno*'s value is printed, you must then refer to *errno*'s entry in the *HP-UX Reference* to find out what the number means.

Another approach is to check for specific values of *errno* and execute different error routines depending on that value. Only a limited number of situations can cause the failure of a particular routine; thus, a routine usually has a small set of values that it can assign to *errno*. To find out what this set is, refer to the routine's entry in the *HP-UX Reference*.

For example, in the *HP-UX Reference* you find that *errno* is set to *ENOENT* (defined in the *errno.h* include file) when you try to open a file that doesn't exist. Once this is known, you can incorporate the following code into the program:

```
#include <errno.h>
#include <fcntl.h>
extern int errno;
main()
{
    int eid;
    if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1)
    {
        if (errno == ENOENT)
            printf("Error occurred because file doesn't exist to open");
        else
            printf("File exists to open, but still an error occurred");
        exit(1);
    }
    ...
}
```

Notice the print statements in the example above could be replaced with calls to more complicated error handling routines such as *perror(3)* (see *HP-UX Reference*).

Resetting Interfaces

The DIL routine for resetting an interface card your program is accessing through an open interface file is *io_reset*. This routine is used on HP-IB or GPIO interfaces.

If the entity identifier for a previously opened interface file associated with the interface is *eid*, the following code resets the interface:

```
io_reset(eid);
```

For a HP-IB interface, resetting involves pulsing its Interface Clear line (IFC); for a GPIO interface the Peripheral Reset line (PRESET) is pulsed. The routine also causes the interface to self-test. If it fails its test, the routine returns a *-1*. If the interface successfully resets and completes its self-test, the routine returns a *0*.

Assume that after opening an interface file you want to make sure the interface operates correctly. This is done by calling *io_reset* and looking at its return value:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open( "/dev/raw_hpib", O_RDWR);
    if (io_reset(eid) == -1)
    {
        printf("Possible problem with interface");
        exit(1);
    }
    ...
}
/*program continues if "io_reset" was successful*/
```

Controlling I/O Parameters

The Device I/O Library provides four routines that allow you to control three different parameters involved in data transfers between an interface card and the devices connected to it. The routines and the parameters that they control are listed below.

Routine	I/O Parameter
<i>io_timeout_ctl</i>	Timeout: Assign a timeout value for data transfers.
<i>io_width_ctl</i>	Data Path Width: Specify width of the interface's data path.
<i>io_speed_ctl</i>	Transfer Speed: Request a minimum speed for data transfers through the interface.
<i>io_eol_ctl</i>	Read Termination Character: Assign a character to be recognized as a read termination character.

When you use one of these four routines, its effect is associated with the open interface file for the interface. If you close the file the effect is lost and the I/O parameter returns to its default state the next time the file is opened.

Setting the I/O Timeout

The I/O timeout parameter controls how long an interface spends trying to complete a data transfer with a device connected to it. When you open the interface file associated with the interface, the timeout is set at 0 by default, indicating that the system never causes a timeout.

If timeout is zero and an error condition occurs which keeps a data transfer from completing, your program will hang. It is recommended you set a timeout for the interface. To set or change the timeout use *io_timeout_ctl*:

```
#include <fcntl.h>
main()
{
    int eid;
    long time;
    eid = open( "/dev/raw_hplib", O_RDWR);
    time = 1000000; /*set timeout of 1 second*/
    io_timeout_ctl(eid, time);
    ...
    ... /*data transfers using "eid" are controlled by the
    ... timeout value "time"*/
}
```

eid is the entity identifier for the open interface file and *time* is a 32-bit long integer specifying the length of the timeout in microseconds.

If *read* or *write* requests do not complete within the time limit specified by the timeout value, the requests are aborted and an error indication is returned (a return value of -1). If a routine fails due to the timeout occurring, *errno* is set to **EIO** (not to be confused with EOI).

Although you specify the timeout value in microseconds when you call *io_timeout_ctl*, the resolution of the effective timeout is system-dependent. The timeout value is rounded up to your system's time resolution boundary. For example, if your system's resolution is 10 milliseconds and you request a timeout of 25000 microseconds (25 milliseconds), the effective timeout is set at 30 milliseconds. Since the timeout value is always rounded up to the nearest time resolution boundary, it is impossible to have a timeout of zero. The smallest timeout you can have is determined by your system's resolution. For example, if the system has a resolution of 10 milliseconds you can set the minimum timeout of 10 milliseconds by specifying a value between zero and 10 milliseconds.

NOTE

Specifying a timeout of zero sets an infinite timeout; the system will never cause a timeout. **Specifying a timeout of zero is not recommended.**

If your program has used *open* more than once to open the same interface file, the entity identifiers returned by *open* can each have their own timeout associated with them. Using *io_timeout_ctl* with one entity identifier does not affect the other entity identifiers.

An entity identifier for an interface file obtained with the HP-UX routine *dup* or inherited by a *fork* request shares the same timeout as the original entity identifier for the file obtained with *open*. If the child process resulting from a *fork* inherits an entity identifier and then changes the timeout, the entity identifier used by the parent process is also affected.

Setting Data Path Width

When you create an interface file and then open it for the first time, the data path width defaults to 8 bits. Once the file is opened, *io_width_ctl* lets you select a new width. **Allowable widths are system and hardware dependent.** You should refer to the Appendix associated with your system to find out what widths are allowed for various interfaces.

Assuming that the open interface file has the entity identifier *eid*, *io_width_ctl* is called with:

```
int  eid, width;
...
...
io_width_ctl(eid, width);
```

where *width* is the number of bits that are in the new data path. The routine returns a *-1* to indicate an error if the width that you specify is not supported on the specified interface.

For example, to change the width of a GPIO data bus from 8 to 16 bits you can use:

```
#include <fcntl.h>
main()
{
    int eid, width;
    width = 16;                /*width of new data path */

    eid = open("/dev/raw_gpio", O_RDWR);
    io_width_ctl( eid, width); /*assign new width for GPIO bus*/
    ...
    ... /*data transfers using "/dev/raw_gpio" will now
    ... use a 16-bit bus*/
}
```

Changing the data path width of an interface with this routine affects all users of the interface. Once you change the data path width, it stays at the new value for each future opening of the file. Either *io_reset* or *io_width_ctl* can be used to change the path back to the default of 8 bits.

Setting Transfer Speed

You can set the minimum transfer speed that is used on the interface (within the limits of the hardware) with the routine *io_speed_ctl*:

```
io_speed_ctl( eid, speed);
```

where *eid* is the entity identifier for the open interface file and *speed* is an integer indicating a minimum speed in *k*-bytes per second (*k* is equal to 1024). The routine returns a 0 if it is successful, and a -1 is an error occurred. For example:

```
io_speed_ctl( eid, 1);
```

requests a minimum speed of 1024 bytes per second. The system may use a faster transfer rate, but you are at least supplied with that speed.

The transfer method (e.g. **DMA**, fast-handshake) chosen by your system is determined by the minimum speed that you request. **DMA** (Direct Memory Access) is the direct transfer of data between memory and I/O interfaces. If you specify a speed that requires a DMA transfer, the system waits until a DMA path is available.

The system selects a transfer method that is as fast or faster than the speed you requested. If you request a speed that is beyond the limitations of the system, the fastest transfer method possible is used. See the hardware-dependency Appendix for specifics.

Setting Read Termination Character

When you perform read operations on an open interface file, certain conditions cause the interface to recognize the end of data transfer from a sending device. When you call *read*, you must specify how many bytes you expect to read. After the specified number of bytes has been read, the data transfer halts. Also, the interface you are accessing can be configured to recognize a special read termination condition. For instance, if an HP-IB interface sees the EOI line asserted, it knows that it has received the last data byte in the transfer and the read operation halts, whether or not the specified byte count has been reached. Similarly, a read operation with a GPIO interface halts when the PSTS line is asserted.

The DIL routine *io_eol_ctl* enables you to set an interface to recognize a particular character as a **read termination character**, in addition to any other termination conditions already in effect for the interface. The call to the routine has the form:

```
int  eid, flag, match;
...
...
io_eol_ctl(eid, flag, match);
```

where *eid* is the entity identifier for the open interface file and *flag* either enables or disables the interface's ability to recognize a special read termination character. When *flag* indicates enable mode and the interface's data path is 8 bits, the least significant byte of *match* is the integer equivalent of the termination character that you want to set. A flag of 0 disables any special read termination character that you have previously set. If the flag has any other value, then the match value indicates a new termination character.

Note that if any special read termination condition defined for the interface is still in effect (e.g. EOI for an HP-IB). Either it or the termination character that you have defined could cause a read operation to halt. Also note the read termination character you set up is interpreted by the interface as the last byte of data. In other words, the interface sees it as part of the data message but does not try to read past it.

If the data path for the interface is set at 16 bits (such as with a GPIO interface), then for most systems the **termination character** is also 16 bits. It is taken from the 2 least significant bytes of the specified *match* value.

To illustrate using *io_eol_ctl*, assume that you want to set up an HP-IB interface to recognize a backslash-n ("*\n*") as a read termination character. First, you must open the HP-IB interface file and obtain the entity identifier *eid*. Second, make the call to *io_eol_ctl* in your program using *eid* as the entity identifier, *ENABLE* as the flag, and "*\n*" as the match:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open("/dev/raw_hpib", O_RDWR);
    io_eol_ctl(eid, ENABLE, "\n");
    ...
    ...          /*data transfers using "eid" terminate with a "\n"*/
    ...
}
```


Now when data is read from `/dev/raw_hpib`, the read operation is terminated when any one of the following occurs:

- The byte count specified in the call to `read` is reached.
- The HP-IB's EOI line is asserted. The character on the bus, when the interface sees the line's assertion, becomes the last byte in the data message.
- A backslash-n ("`\n`") is read. The backslash-n ("`\n`") becomes the last byte in the data message.

If your program has used `open` more than once to open the same interface file, the entity identifiers returned by `open` can each have their own read termination character associated with them. Using `io_eol_ctl` with one entity identifier does not effect the others. Thus, you can set up several entity identifiers for the same interface that recognize different termination characters.

An entity identifier for an interface file obtained with the HP-UX system routine `dup` or inherited by a `fork` request shares the same read termination character as the original entity identifier. If the child process resulting from a `fork` inherits an entity identifier and then sets a read termination character for it, the entity identifier used by the parent process is also affected.

Removing a Read Termination Character

There are two ways that your program can disable an interface from interpreting a read termination character that the program has previously set.

1. Close the interface file and then reopen it. The new entity identifier for the file will not know about the termination character.
2. Disable the termination character by calling `io_eol_ctl` with a flag of 0:

```
io_eol_ctl(eid, 0, XX);
```

The `XX` indicates a **don't care** value for the match argument. If the flag is 0, then the match value is not looked at by the routine.

The code below sets up the ASCII "`.`" (46) as a termination character, does a read operation, and then disables the termination character.

```
#include <fcntl.h>
main()
{
    int eid;
    char buffer[12];
    eid = open("/dev/raw_hpib", O_RDWR);
    io_eol_ctl(eid, 1, 46);
    read( eid, buffer, 12); /*Read operation halts when either a
                           ". " is read or when the 12th byte is read*/
    io_eol_ctl( eid, 0, 0); /*termination character is removed*/
    ...
    ...
}
```

Determining Why a Read Terminated

There are several situations which can terminate read operations through an interface. After your program completes a *read*, you may want to include code that makes sure the cause of the *read*'s termination is what you expected. The DIL routine that allows you to do this is *io_get_term_reason*.

io_get_term_reason accepts the entity identifier of the interface file as an argument and returns an integer. The returned value indicates how the last read operation ended, as shown below.

Returned Value	Meaning
-1	An error occurred while making this routine call.
0	The last read terminated abnormally (for some reason other than the ones covered below).
1	The last read terminated by reading the number of bytes requested.
2	The last read terminated by detecting a previously determined read termination character.
4	The last read terminated by detecting some device-imposed termination condition. Examples are the assertion of EOI for an HP-IB, the assertion of PSTS for a GPIO, or an end-of-record mark on a 9-track tape.

If a read terminated for multiple reasons, the bits that are set indicate each of the reasons. The three least significant bits of the lowest byte have the meanings indicated by their associated decimal values in the table above. For example, if *io_get_term_reason* returns a 7 you know that the specified number of bytes were read, the last byte read was a read termination character, and also a device-defined termination condition occurred.

NOTE

If no *read* is performed on an interface file once it is opened and you call *io_get_term_reason*, the routine returns a 0.

All entity identifiers descending from one *open* request (such as from *dup* or *fork*) affect the status returned by this routine. For example, suppose that an entity identifier is inherited by a child process through a *fork*. If the parent process calls *io_get_term_reason*, the last read operation of either the parent or the child is looked at, depending on which is more recent.

An Example

Assume that your system is a Series 500 and that you want to read data from a device on an HP-IB and need to guarantee that a specific number of bytes are read. The following code reads 50 bytes through an opened interface file and makes sure that *read* wasn't terminated before all 50 were read.

```

#include <fcntl.h>
main()
{
    int eid, condition;
    char buffer[50];    /*storage for data*/

    eid = open("/dev/raw_hpib", O_RDWR);
    read(eid, buffer, 50); /*perform read and put data in "buffer"*/
    if ((condition = io_get_term_reason(eid)) > 1)
        /*Terminated due to seeing a read termination character or the
        assertion of EOI. However, the event could have occurred at the
        same time as the 50th byte was read*/
        printf("Possible termination before all of data was read");

    else if (condition < 1)
    {
        if (condition == 0)
            /*Termination due to some abnormal condition*/
            printf ("Last read terminated abnormally");

        else
            printf ("io_get_term_reason call failed");
    }
    else
        /*Termination due to reading the 50th byte*/
        printf("All of data was read into buffer");
}

```

Note that on the Series 500, the value returned by *io_get_term_reason* only indicates the termination cause with the highest value; other causes with lower values could have occurred at the same time. See *Appendix A: Series 500 Dependencies* for more information.

Interrupts

DIL provides an **interrupt** mechanism that is similar to HP-UX signal handling. The user is able to set up **interrupt handlers** to be invoked when certain conditions occur. DIL currently supports interrupts for HP-IB and GPIO interfaces.

The following interrupt conditions are available for HP-IB interfaces:

Name	Meaning
SRQ	SRQ line has been asserted
TLK	The computer has been addressed to talk
LTN	The computer has been addressed to listen
TCT	The computer has received control of the bus
IFC	The IFC line has been asserted
REN	The remote enable line has been asserted
DCL	The computer has received a device clear command
GET	The computer has received a group execution trigger command
PPOLL	A specific parallel poll response occurred

The following interrupt conditions are available for the GPIO interface:

SIE0	Status line 0 has been asserted
SIE1	Status line 1 has been asserted

Check the hardware-dependency Appendix for your system for any restrictions that may apply.

io_on_interrupt

DIL provides two routines for controlling interrupts. The first routine, *io_on_interrupt*, sets up the interrupt information and has the form:

```
io_on_interrupt(eid, cause_vec, handler);
```

where *eid* is an entity identifier for a GPIO or raw HP-IB interface. The parameter *handler* points to a function to be invoked when the condition occurs. Then *cause_vec* is a pointer to a structure of the form:

```
struct interrupt_struct {
    int cause;
    int mask;
};
```

The *interrupt_struct* structure is defined in the include file *dvio.h*.

The *cause* parameter is a bit vector specifying which of the interrupt or fault events will cause the *handler* routine to be invoked. The interrupt *causes* are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the *io_on_interrupt* capability. Specifying a zero-valued *cause* vector effectively turns off the interrupt for that *eid*.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. The integer *mask* specifies which parallel poll response lines are of interest. *mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to one of the eight lines. For example, if you want an interrupt handler invoked for a response on lines 2 or 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask*.

Upon occurrence of an enabled interrupt condition on the specified *eid*, the receiving process executes the interrupt-handler routine pointed to by *handler*. The entity identifier *eid* and the interrupt condition *cause* are returned as the first and second parameters respectively.

An interrupt for a given *eid* is implicitly disabled after the event occurs. The interrupt condition can be re-enabled with *io_interrupt_ctl(3)*.

io_on_interrupt returns a pointer to the previous handler if the new handler is successfully installed, otherwise it returns a -1 and *errno* is set.

The following example illustrates how an interrupt handler can be set up to handle assertion of the service request line (SRQ):

```
#include <dvio.h>
#include <fcntl.h>
#include <stdio.h>
main()
{
    int eid;
    struct interrupt_struct cause_vec;
    eid = open ("/dev/raw_hpib", O_RDWR);
    cause_vec.cause = SRQ;
    io_on_interrupt(eid, cause_vec, handler);
    ...
}
handler (eid, cause_vec);
int eid;
struct interrupt_struct cause_vec;
{
    if (cause_vec.cause == SRQ)
        service_routine(); /* user specific routine*/
}
```

io_interrupt_ctl

The *io_interrupt_ctl(3D)* routine allows the user to enable or disable interrupts on a specific *eid*. Since interrupts are automatically disabled when an interrupt occurs, *io_interrupt_ctl* is commonly used when the user wants to repeatedly handle a specific event. *io_interrupt_ctl* has the following form:

```
io_interrupt_ctl(eid, enable_flag);
```

where *eid* is an entity identifier for an open GPIO or raw HP-IB device file. To control enabling and disabling of the interrupts, *enable_flag* is used. If *enable_flag* is non-zero, then interrupts are enabled on the *eid*. If *enable_flag* is zero, then interrupts are disabled on the *eid*. Note that attempting to use *io_interrupt_ctl* on an *eid* that has not had an *io_on_interrupt* applied to it, fails.

The following example modifies the handler from the previous example to re-enable interrupts:

```
handler(eid, cause_vec);
int eid;
struct interrupt_struct cause_vec;
{
    if (cause_vec.cause == SRQ)
    {
        service routine(); /* user specific routine*/
        io_interrupt_ctl(eid,1);
    }
}
```



To gain a full range of control over your computer's HP-IB interface you must use:

- the general purpose I/O routines in DIL discussed in *Chapter 2: General Purpose Routines*
- the DIL routines designed specifically for controlling the HP-IB interface that are described in this chapter

Besides the various routines, you must know about the commands that are interpreted on an HP-IB. This chapter provides some general information about HP-IB commands and introduces the DIL routines that specifically control the HP-IB. Then it relates this information to the information provided in *Chapter 2: General Purpose Routines* to illustrate some HP-IB interfacing strategies.

Overview of HP-IB Commands

This section discusses the HP-IB commands that are sent over the 8 data lines while the ATN line is asserted. You can send all of these commands using a DIL routine called *hpib_send_cmnd*. This routine takes care of the assertion of ATN and the necessary handshaking between devices. The computer's interface must be the Active Controller before *hpib_send_cmnd* is used and any of the HP-IB commands sent. How *hpib_send_cmnd* is called from your program is discussed later in this chapter.

In order for the commands to be interpreted by devices on the HP-IB, the bus's remote enable line (REN) must be in its enabled state. Only the System Controller changes the state of this line (see the *System Controller's Duties* section later in this chapter). By default, REN is enabled.

Commands sent on the bus's data line form 4 groups:

- **Universal commands** cause every device, so equipped, to perform a specific interface operation. The devices do not have to be addressed as listeners.
- **Addressed commands** are similar to the universal commands, except they affect only those devices currently addressed as listeners.
- **Talk and listen addresses** are commands that assign talkers and listeners on the bus.
- **Secondary commands** are commands that must always be used in conjunction with a command from one of the above groups.

The table below lists the commands that you can send with *hpib_send_cmnd*. Later, when you use the routine, you may need to refer back to this table for the decimal or ASCII character value of particular commands.

Table 3.1 Bus Commands

Command	Decimal Value	ASCII Character
Universal Commands:		
UNLISTEN	63	?
UNTALK	95	-
DEVICE CLEAR	20	DC4
LOCAL LOCKOUT	17	DC1
SERIAL POLL ENABLE	24	CAN
SERIAL POLL DISABLE	25	EM
PARALLEL POLL UNCONFIGURE	21	NAK
Addressed Commands:		
TRIGGER	8	BS
SELECTED DEVICE CLEAR	4	EOT
GO TO LOCAL	1	SOH
PARALLEL POLL CONFIGURE	5	ENQ
TAKE CONTROL	9	HT
Talk and Listen Addresses:		
Talk Addresses 0-30	64-94	@ thru ^ (uppercase ASCII)
Listen Addresses 0-30	32-62	space thru > (numbers and special characters)
Secondary Commands: (If a secondary command follows the PARALLEL POLL CONFIGURE command then it is interpreted as follows, otherwise its meaning is device-dependent)		
PARALLEL POLL ENABLE	96-111	' thru o (lowercase ASCII)
PARALLEL POLL DISABLE	112	p

UNLISTEN

The UNLISTEN command **unaddresses** all current listeners on the bus. Single listeners cannot be unaddressed without unaddressing all listeners. It is necessary to use this command to guarantee only desired listeners are addressed.

UNTALK

The UNTALK command unaddresses the current talker. Sending an unused talk address accomplishes the same thing. This command is provided for convenience since addressing one talker automatically unaddresses others.

DEVICE CLEAR

The DEVICE CLEAR command causes all **recognizing devices** to return to a pre-defined, device-dependent state. Recognizing devices respond whether or not they are addressed. Device manuals define the reset state for each device that recognizes the command.

LOCAL LOCKOUT

The LOCAL LOCKOUT command disables local control on all devices that recognize this command. Recognizing devices respond to the command whether or not they are addressed.

SERIAL POLL ENABLE

The SERIAL POLL ENABLE command establishes serial poll mode for all responding devices capable of being bus talkers. Recognizing devices respond to the command whether or not they are addressed. When a device is addressed to talk, it returns a 8-bit status byte message.

This command is not discussed any further since its function is accomplished by a DIL routine called *hpib_spoll* (discussed later in this chapter).

SERIAL POLL DISABLE

The SERIAL POLL DISABLE command terminates serial poll mode for all responding devices. Recognizing devices respond to the command whether or not they are addressed.

This command is not discussed any further since its function is accomplished by a DIL routine called *hpib_spoll* (discussed later in this chapter).

TRIGGER (Group Execute Trigger)

The TRIGGER command causes the devices that are currently addressed as listeners to initiate a preprogrammed, device-dependent action if they are capable. Device manuals indicate whether or not a particular device is capable of responding to the TRIGGER command and if it can, how to program it to do so.

SELECTED DEVICE CLEAR

The SELECTED DEVICE CLEAR command resets devices currently addressed as listeners to a device-dependent state, if they are capable. A device's documentation indicates whether or not the device recognizes this command and if so, it defines the reset state.

GO TO LOCAL

The GO TO LOCAL command causes devices that are currently addressed as listeners to return to the local control state (exit from the remote state). The devices return to the remote state the next time they are addressed.

PARALLEL POLL CONFIGURE

The PARALLEL POLL CONFIGURE command tells the devices currently addressed as listeners that a secondary command follows. This secondary command must be either PARALLEL POLL ENABLE or PARALLEL POLL DISABLE.

PARALLEL POLL ENABLE

The PARALLEL POLL ENABLE command configures devices addressed by the PARALLEL POLL CONFIGURE command to respond to parallel polls on a particular data line and with a particular logic level. Some devices implement a local form of this message (for example, jumpers) that cannot be changed.

This command must be preceded by the PARALLEL POLL CONFIGURE command.

PARALLEL POLL DISABLE

The PARALLEL POLL DISABLE command disables devices addressed by the PARALLEL POLL CONFIGURE command from responding to parallel polls. This command must be preceded by the PARALLEL POLL CONFIGURE command.

Overview of the HP-IB DIL Routines

Besides the general purpose routines described in *Chapter 2: General Purpose Routines*, DIL also provides routines that allow you to fully access the capabilities of the HP-IB interface. There are 14 of these routines:

<i>hpib_abort</i>	Stops activity on a specified HP-IB select code.
<i>hpib_io</i>	Performs a mixture of HP-IB read and write activities.
<i>hpib_ppoll</i>	Conducts parallel poll on HP-IB.
<i>hpib_spoll</i>	Conducts serial poll on HP-IB.
<i>hpib_bus_status</i>	Returns status on HP-IB interface.
<i>hpib_eoi_ctl</i>	Controls EOI mode for data transfers.
<i>hpib_pass_ctl</i>	Changes active controllers on HP-IB.
<i>hpib_card_ppoll_resp</i>	Configures it owns response to a parallel poll.
<i>hpib_ren_ctl</i>	Controls remote enable line (REN) on HP-IB.
<i>hpib_rqst_srvc</i>	Allows interface to generate an SRQ request on HP-IB.
<i>hpib_send_cmnd</i>	Sends characters on HP-IB with the attention line (ATN) line asserted.
<i>hpib_wait_on_ppoll</i>	Lets you wait for a particular parallel poll value to occur.
<i>hpib_status_wait</i>	Lets you wait until a particular status condition is true.
<i>hpib_ppoll_resp_ctl</i>	Defines interface parrallel poll response as yes or no.

The Computer's Role on the HP-IB

Your computer must currently have one of the following two roles on the HP-IB:

- It is the **Active Controller**.
- If it isn't the Active Controller, it is a **Non-Active Controller**.

There can be only one Active Controller on a HP-IB interface at a given time. Since Active Controller status is passed between bus controller devices, your computer's status can change from **active** to **non-active** or from **non-active** to **active**.

In addition to being either an Active or Non-Active Controller, your computer can also be the bus's **System Controller**. Once a controller is configured as the System Controller, it cannot be unconfigured without powering down the system. The System Controller is either the Active Controller or a Non-Active Controller. When the System Controller is initially powered up, it assumes the role of Active Controller.

Which of the DIL routines you can use depends on your computer's role on the HP-IB. Given the three role designations, the table below (*Table 3.2*) indicates which routines can be used with them.

Table 3.2 DIL Routine Role Designations

Routine	System Controller	Active Controller	Non-Active Controller
hpib_abort	X		
hpib_io		X	
hpib_ppoll		X	
hpib_spoll		X	
hpib_bus_status	(X)	X	X
hpib_eoi_ctl	X		
hpib_pass_ctl		X	
hpib_card_ppoll_resp		X*	X
hpib_ren_ctl	X		
hpib_rqst_srvc		X*	X
hpib_send_cmnd		X	
hpib_wait_on_ppoll		X	
hpib_status_wait	(X)	X	X
hpib_ppoll_resp_ctl		X*	X

* means that the routine can be used if the computer is the Active Controller but there is no effect until it becomes a Non-Active Controller.

(X) means that the X isn't required since the System Controller must be either **active** or **non-active** and both of these roles can use the routine (i.e. the **System Controller** role is not required to use the routine).

Opening the HP-IB Interface File

Chapter 2: General Purpose Routines discusses how the interface file for the HP-IB must be created so that the DIL routines can be used and this chapter discusses how it is opened.

The following code indicates how to open a raw HP-IB interface file called */dev/hpib*:

```
int eid;
eid = open("/dev/hpib", O_RDWR);
```

eid is the entity identifier for the opened file and it is required when you want to specify the file from program processes. The *O_RDWR* indicates that you want read and write access to the interface file.

The code above does not check whether or not the file was opened successfully. To verify that no errors occurred, your program should contain an error check when the file is opened:

```
...
...
if (( eid = open("/dev/hpib",O_RDWR)) == -1)
{
    printf("can't open file");
    exit(1)
}
...
...
```

Sending HP-IB Commands

The DIL routine that allows you to place HP-IB commands on the data bus is *hpib_send_cmnd*. Your computer must be the Active Controller to use this routine.

One method of using this routine is to first set up a character array containing the commands that you want to send. You assign the decimal value for the commands to the elements of the array. The routine call then has the form:

```
hpib_send_cmnd( eid, command, number);
```

where *eid* is the entity identifier for the open interface file, *command* is a character pointer to the first element of the array containing the HP-IB commands, and *number* is the number of elements (commands) in the array. The routine *hpib_send_cmnd* places each of the commands stored in the array on the bus with ATN asserted.

Notice that by changing the *number* argument and moving the *command* pointer you can send subsets of command arrays. Suppose you create an array that contains 10 HP-IB commands, *command[0]* through *command[9]*. You can now specify that only the last 5 commands in the array be sent using:

```
hpib_send_cmnd( eid, command + 5, 5);
```

This method of sending HP-IB commands by storing them in an array uses their decimal values. Alternatively, the commands' ASCII character values can be used by specifying a character string. In this case, the routine call has the form:

```
hpib_send_cmnd( eid, "command_string", number);
```

where *eid* and *number* are the same as above. However, the commands to be sent are now specified by each character in the string *command_string*.

To illustrate the two methods, assume that you want to send the HP-IB UNLISTEN and UNTALK commands. With the decimal array method you first set up an array with two elements, the decimal values for the commands, and then call *hpib_send_cmnd*:

```
#include <fcntl.h>
main()
{
    int eid;
    char command[2];          /*command array*/
    eid = open("/dev/raw_hpib", O_RDWR);
    command[0] = 63;         /*decimal value for UNLISTEN*/
    command[1] = 95;        /*decimal value for UNTALK*/
    hpib_send_cmnd( eid, command, 2);
}
```

If the ASCII character string method is used, the same effect is achieved with the code:

```
#include <fcntl.h>
main()
{
    int eid;

    eid = open("/dev/raw_hpib", O_RDWR);
    hpib_send_cmd( eid, "?_", 2); /*? is ASCII for UNLISTEN and*/
                                /*_ is ASCII for UNTALK    */
}

```

Since the array method allows you to store a list of commands, it should be used if you are sending a large number of commands or if you are sending the same set of commands several times in a program. With the string method, the entire set of commands must be specified as a string in the call to *hpib_send_cmd*. It is useful if you are sending only a few commands or if a particular set of commands is only sent once in a program.

Errors While Sending Commands

Normally, *hpib_send_cmd* returns a 0 if it executes successfully. However, it returns a -1 if any one of the following error conditions are true:

- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, the program should check the value of *errno*, an external integer variable used by HP-UX system calls. *Chapter 2: General Purpose Routines* discusses how you can design an error checking routine that looks at the value of *errno*.

The following table indicates the value that *errno* will have given that one of the above conditions occurred during the call to *hpib_send_cmd*:

Errno Value Error Condition

EBADF	<i>eid</i> did not refer to an open file
ENOTTY	<i>eid</i> did not refer to a raw interface file
EIO	The interface was not the Active Controller

The Active Controller

Acting as the Active Controller of the bus involves sending the HP-IB commands with *hpib_send_cmdnd* and making calls to several other DIL routines. The functions of the Active Controller discussed in this chapter are:

- Setting up devices as talkers and listeners
- Gaining remote control of devices
- Locking out local control of devices
- Enabling local control of devices
- Triggering devices to initiate device-dependent actions
- Transferring data
- Clearing devices
- Servicing requests from devices
- Conducting parallel and serial polls
- Passing active control of the bus to another controller

Determining Active Controller

To carry out the Active Controller's bus management activities, the computer's HP-IB interface must be the Active Controller of its bus. If other devices on the bus are capable of being the Active Controller, you can use the *hpib_bus_status* routine to determine if the interface is currently the Active Controller.

To find out if the interface is the Active Controller, the call to *hpib_bus_status* must have the form:

```
hpib_bus_status( eid,4);
```

where *eid* is the entity identifier for the opened HP-IB interface device file and the *4* tells the routine to determine if the interface is the Active Controller. This routine returns a value that can be tested, see source code below.

hpib_bus_status returns 0 if the answer is no, 1 if the answer is yes, and -1 if an error occurred. The code that follows shows a straightforward way of interpreting the returned value:

```
#include <fcntl.h>
main()
{
    int eid, status;
    eid = open("/dev/raw_hpib", O_RDWR);

    if ((status = hpib_bus_status( eid,4)) == -1)
        ... /*an error occurred -- insert code that*/
        ... /*flags it. */
    else if (status == 0)
        ... /*not Active Controller -- insert code */
        ... /*that requests Active Controller status*/
    else
        ... /*is Active Controller -- insert code for*/
        ... /*the bus management routine required */
}
```

Setting Up Talkers and Listeners

One talker and one or more listeners must be configured on the bus before data can be transferred. Also, some HP-IB commands effect only those devices currently addressed as listeners, which means that the Active Controller must specify the listeners before using them. There can be only one talker at a time on the bus, but there can be any number of listeners.

There are two methods for addressing listeners and talkers on an HP-IB. The first method, referred to as **auto-addressing**, instructs the computer to handle addressing for you. The second method requires using the *hpib_send_cmd* function to **manually** address the bus.

The system performs auto-addressing on normal (**non-raw**) HP-IB device files. Note that DIL routines require a raw HP-IB device file. Therefore, while you can *open*, *close*, *read*, and *write* from a non-raw HP-IB device file, the DIL functions will fail.

You can create a device file that informs the system to perform auto-addressing using the *mknod* command (described in *Chapter 2: Creating an Interface File*). The following example creates an HP-IB device file for a specific device on select code 1 at bus address 3. (This assumes we are using a driver of 12 on a Series 500 with an HP27110A/B card at select code 1.):

```
mknod /dev/device c 12 0x010300
```

The following code illustrates auto-addressing using this device file:

```
main()
{
    int eid;
    eid = open("/dev/device",O_RDWR);
    /*Assuming "/dev/device" has the minor number (0x010300), the*/
    /*system addresses the interface card at select code 1 as a talker*/
    /*and the device at bus address 3 as a listener before sending data*/
    write(eid, "test data",9);
}
```

Talkers and listeners may be manually configured with the HP-IB commands formed by the talk and listen addresses of the devices. First, however, you should remove any previous listeners from the bus with the UNLISTEN command. To configure the bus's talker and listeners, the following steps are required:

1. Send the UNLISTEN command to remove any previous listeners.
2. Send the talk address of the device that will be sending data. There can only be one talker device.
3. Send the listen address of each device that is to receive the data.

To send the HP-IB commands necessary for this process you can use the *hpib_send_cmd* routine.

Calculating Talk and Listen Addresses

A talker is specified on the bus by sending the talk address for the device and a device is specified as a listener by sending its listen address. Talk addresses and listen addresses are considered HP-IB commands, which means you should send them with the *hpib_send_cmnd* routine.

To calculate either the talk or the listen address for a device, you must know its HP-IB address. The HP-IB address for the computer's interface card is found with the *hpib_bus_status* routine:

```
#include <fcntl.h>
main()
{
    int eid, address;
    eid = open("/dev/raw_hpib", O_RDWR);
    address = hpib_bus_status( eid, 7);
    ...
}
```

where *eid* is the entity identifier for the interface file and *7* indicates that you want the routine to return the interface's HP-IB address. To find out the bus address of some other device, refer to its installation and operation documentation.

Once you have the device's HP-IB address, its *talk_address* (in decimal) is found with the formula:

$$\text{talk_address} = 64 + \text{bus_address}$$

where *bus_address* is the HP-IB bus address for the device. Bus addresses range from 0 to 30. The listen address for a device (in decimal) is found similarly with the formula:

$$\text{listen_address} = 32 + \text{bus_address}$$

Thus, **My Talk Address (MTA)** for the computer is calculated with:

$$\text{MTA} = \text{hpib_bus_status}(\text{eid}, 7) + 64;$$

and **My Listen Address (MLA)** is calculated with:

$$\text{MLA} = \text{hpib_bus_status}(\text{eid}, 7) + 32;$$

An Example Configuration

Assuming that the computer's interface is currently the Active Controller of the HP-IB, the following code establishes the interface as the bus talker. Two devices at HP-IB addresses 4 and 8 are designated as the bus listeners.

```
#include <fcntl.h>
main()
{
    int eid, MTA;
    char command[4];
    eid = open("/dev/raw_hpib", O_RDWR);
    MTA = hpib_bus_status( eid, 7) + 64; /*calculate My Talk Address*/
    command[0] = 63; /* the UNLISTEN command*/
    command[1] = MTA; /* the talk address for the interface*/
    command[2] = 32 + 4; /* the listen address for device at HP-IB address 4*/
    command[3] = 32 + 8; /* the listen address for device at HP-IB address 8*/
    hpib_send_cmnd( eid, command, 4);
}
```

Remote Control of Devices

Most HP-IB devices can be controlled either from their front panel or from the bus. If the device's front-panel controls are currently operational, it is in the **local state**. If it is being controlled through the HP-IB, it is in its **remote state**. Pressing the device's front-panel **LOCAL** key returns the device to local control, unless it is in the local lockout state (described in a subsequent section).

The level of the remote enable (REN) line of the HP-IB bus controls whether or not a device can respond to remote program control. If the REN line is enabled, any device that is addressed (as either a talker or a listener) is automatically placed in the remote state. Only the System Controller can change the level of the REN line (see *System Controller's Duties* later in this chapter). By default, the line is enabled when the System Controller is powered up.

Locking Out Local Control

The LOCAL LOCKOUT command effectively locks out the **local** switch present on most HP-IB front panels, preventing a device's user from interfering with the system operations by pressing buttons. All devices that recognize this command are affected, whether they are addressed or not, and cannot be returned to local control from their front panels.

The following code shows one way of sending the LOCAL LOCKOUT command:

```
...
...
command[0] = 17;          /* Decimal value of LOCAL LOCKOUT*/
hpib_send_cmdnd( eid, command, 1);
...
...
```

The local lockout state is cancelled by sending a GO TO LOCAL command to a device.

Enabling Local Control

During system operation, it may be necessary for an operator to interact with one or more devices in the local state. For instance, an operator might need to work from the front panel to make special tests or to troubleshoot. The GO TO LOCAL command returns all of the devices currently addressed as listeners to the local state.

For example, the code below places the devices at HP-IB addresses 3 and 5 into their local state.

```
...
command[0] = 63;          /* the UNLISTEN command*/
command[1] = 32 + 3;      /* listen address for device at address 3*/
command[2] = 32 + 5;      /* listen address for device at address 5*/
command[3] = 1;          /* the GO TO LOCAL command*/
hpib_send_cmdnd( eid, command, 4);
...
...
```

Triggering Devices

The HP-IB TRIGGER command tells the devices currently addressed as listeners to initiate some device-dependent action. For example, it can be used to trigger a digital voltmeter to perform its measurement cycle. Because the response of a device to a TRIGGER command is strictly device-dependent, you can not specify with the command what action is to be initiated.

The following code triggers the device at bus address 5 to initiate some action:

```
...
command[0] = 63;          /* the UNLISTEN command*/
command[1] = 32 + 5;     /* the listen address for device at*/
                        /* address 5 */
command[2] = 8;         /* the TRIGGER command*/
hpib_send_cmdnd( eid, command, 3);
...
```

Transferring Data

For the Active Controller to send data to another device it must:

1. Send an UNLISTEN command.
2. Send its own talk address (MTA).
3. Send the listen address of the device that is to receive the data. One listen address is sent for every device that is to receive the data.
4. Send the data.

The first 3 steps are accomplished using *hpib_send_cmdnd*, while the system routine *write* takes care of the fourth.

The following code illustrates how character data can be sent to a device at HP-IB address 5.

```
#include <fcntl.h>
main()
{
    int eid, MTA;
    char command[50];

    eid = open("/dev/raw_hpib", O_RDWR);
    MTA = hpib_bus_status( eid, 7) + 64; /*calculate MTA*/
    command[0] = 63;                    /*the UNLISTEN command*/
    command[1] = MTA;                   /*talk address of interface*/
    command[2] = 32 + 5;                 /*listen address of device at*/
                                        /*address 5 */
    hpib_send_cmdnd( eid, command, 3);
    write( eid, "data message", 12); /*send the data*/
}
```

Now assume that you are expecting to receive 50 bytes of data from another device on the bus. The code below allows the interface to receive character data from a device at bus address 5. The integer variable *MLA* contains the bus address of the interface.

```
#include <fcntl.h>
main()
{
    int eid, MLA;
    char buffer[50];           /*storage for data*/

    eid = open("/dev/raw_hpib", O_RDWR);
    MLA = hpib_bus_status( eid, 7) + 32; /*calculate MLA*/
    command[0] = 63;           /*the UNLISTEN command*/
    command[1] = 64 + 5;      /*the talk address of device at*/
                                /*address 5 */
    command[2] = MLA;         /*the listen address of interface*/
    hpib_send_cmnd( eid, command, 3);
    read( eid, buffer, 50);   /*store the data in "buffer"*/
    printf("Data read is: %s", buffer); /*print message*/
}
```

Clearing HP-IB Devices

There are two HP-IB commands for resetting devices to their pre-defined, device-dependent states. The first one is the DEVICE CLEAR command which causes all devices that recognize the command to be reset, whether they are addressed or not.

Thus, to reset all of the devices on an HP-IB accessed through a interface file with an entity identifier *eid*, you can use the following code:

```
command[0] = 20;             /* the DEVICE CLEAR command*/
hpib_send_cmnd( eid, command, 1);
```

The second command for resetting devices is SELECTED DEVICE CLEAR. This command resets only those devices that are currently addressed as listeners.

To reset a device with an HP-IB address of 7, you can use the following code:

```
...
command[0] = 63;             /* the UNLISTEN command*/
command[1] = 32 + 7;         /* the listen address for device at*/
                                /* address 7 */
command[2] = 4;             /* the SELECTED DEVICE CLEAR command*/
hpib_send_cmnd( eid, command, 3);
...
```

Servicing Requests

Most HP-IB devices, such as voltmeters, frequency counters, and spectrum analyzers, are capable of generating a **service request** when they require the Active Controller to take some action. **Service requests** are generally made after the device has completed a task (such as taking a measurement) or when an error condition exists (such as a printer being out of paper). The operating or programming manual for each device describes the device's capability to request service and the conditions under which it requests service.

Seeing the SRQ Line

To request service, a device asserts the Service Request (SRQ) line on the bus. To determine if SRQ is being asserted, you check the status of the line, wait for SRQ, or set up an interrupt handler for SRQ. The *hpib_status_wait* routine allows you to write code that waits until the SRQ line is asserted before it continues. To specify that you want the program to wait until the SRQ line is asserted, *hpib_status_wait* must be invoked as follows:

```
hpib_status_wait( eid, 1);
```

where *eid* is the entity identifier for the open interface file and *1* indicates that the event that you are waiting for is the assertion of the SRQ line. The routine returns 0 when the condition requested becomes true or -1 if a timeout or an error occurred. This code illustrates *hpib_status_wait*:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open("/dev/raw_hpib", O_RDWR);
    io_timeout_ctl(eid,10000000);
    if (hpib_status_wait( eid, 1) == 0)
        service_routine();           /*SRQ is asserted; service the request*/
    else
        printf("Either a timeout or an error occurred");
}
```

Another solution is to periodically check the value of the SRQ line with *hpib_bus_status*. To check the SRQ line with *hpib_bus_status*, the call looks like this:

```
hpib_bus_status( eid, 1);
```

where *eid* is the entity identifier for the open interface file and *1* indicates that you want the logical value of the SRQ line returned. The routine returns 1 if SRQ is asserted, 0 if it isn't, and -1 if an error occurred.

The most practical way to monitor the SRQ line is to set up an interrupt handler for that condition (see *Chapter 2: Interrupts* section).

The Service Routine

Once a device has asserted the SRQ line, it continues to assert the line until its request has been satisfied. How a service request is satisfied is device-dependent. Serial polling the device can provide the information as to what kind of service it requires.

In many cases, devices requesting service **clear** the SRQ line when they are serially polled. They see the poll as an acknowledgement from the Active Controller to the device that the request has been seen and the Active Controller is responding.

If there is more than one device on the bus and the SRQ line is asserted, any one of the devices could be asserting the line. The Active Controller must then determine which of the devices needs service. There are two strategies for doing this:

- Serial poll each device until you find the one that is requesting service. This approach is reasonable if there are only a few devices on the bus.
- Conduct a parallel poll to locate the device requesting service. Normally, each device (that is capable) is programmed to respond on a different data line. However, since there can be 15 devices on the bus and there are only 8 data lines, it is sometimes necessary to have several devices respond on the same line.

If several devices are programmed to respond on the same parallel poll line and the parallel poll shows that line asserted, the Active Controller must then serially poll each of these devices until it finds the one that is requesting service.

Thus, the Active Controller usually takes one of two approaches in response to seeing the SRQ line asserted: it can conduct a serial poll or it can conduct a parallel poll. In some cases the Active Controller may need to take both types of polls. The DIL routines that conduct these polls are *hpib_ppoll* and *hpib_spoll*. How these routines are used is discussed next.

Parallel Polling

The parallel poll is the fastest means of gathering device status when several devices are connected to the bus. Each device (with this capability) can be programmed to respond with one bit of status when parallel polled, making it possible to obtain the status of several devices in one operation. If a device responds affirmatively (**I need service**) to the parallel poll, more information as to its specific status can be obtained by conducting a serial poll of the device.

Configuring Parallel Poll Responses

Certain devices can be remotely programmed by the Active Controller to respond to a parallel poll. However, other devices require that the response be configured locally. Refer to the documentation for the device whose response you want to configure to find out if remote configuration by the Active Controller is possible.

The Active Controller remotely configures a device's parallel poll response by sending the HP-IB command PARALLEL POLL CONFIGURE followed by PARALLEL POLL ENABLE. The combination of these two commands tells devices addressed as listeners to respond to any future parallel polls on a particular data line and with a particular logic level. Some devices may implement a local form of this message (for example, jumpers) that can not be changed remotely by the Active Controller.

There are 16 different PARALLEL POLL ENABLE commands, each configuring a response on a specific data line and at a specific level. The 8-bits of the command have the following binary form:

D7	D6	D5	D4	D3	D2	D1	D0	Decimal Range:
0	1	1	0	L	X	X	X	96-111

where:

- L** indicates the logic sense of the response (e.g. *1* means that the device will respond with 1 when it needs service)
- X** indicates the data line on which the device will respond

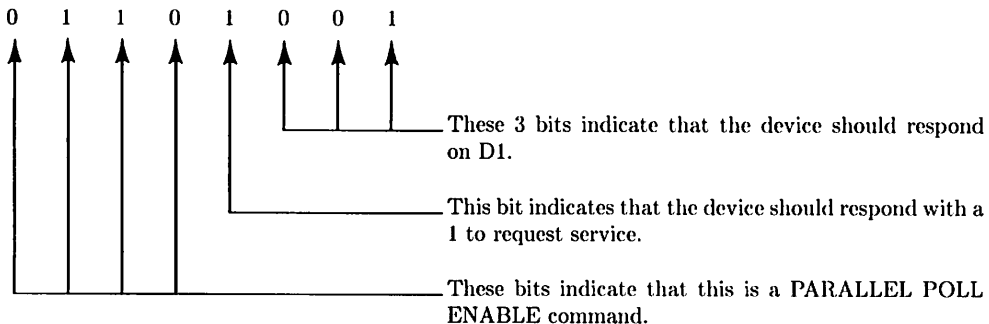
For example, given that the parallel response lines are labeled D0 to D7, a PARALLEL POLL ENABLE command with a decimal value of 104 (01101000 in binary) tells the addressed device to respond to parallel polls on data line D0 with a 1 when it needs service.

The following code shows how you can configure a device at bus address 5 to respond to a parallel poll by asserting data line D1 high when it needs service.

```
#include <fcntl.h>
main()
{
    int eid, MTA;
    char command[50];

    eid = open("/dev/raw_hpib", O_RDWR);
    MTA = hpib_bus_status( eid, 7) + 64; /*calculate MTA*/
    command[0] = MTA; /*talk address of interface*/
    command[1] = 63; /* the UNLISTEN command*/
    command[2] = 32 + 5; /* the listen address for device at*/
    /* address 5 */
    command[3] = 5; /* the PARALLEL POLL CONFIGURE command*/
    command[4] = 105; /* the PARALLEL POLL ENABLE command*/
    hpib_send_cmdnd( eid, command, 5);
}
```

Notice that the bit pattern for the PARALLEL POLL ENABLE command 105 used above is:



When the interface is the Active Controller, it can configure its own parallel poll response by addressing itself as both the talker and the listener. However, the configuration has no effect until the interface is no longer the Active Controller. The Active Controller never responds to parallel polls.

Disabling Parallel Poll Responses

A device whose parallel poll response can be remotely configured by the Active Controller can also be disabled from responding.

The Active Controller disables a device from responding to any future parallel polls by first sending a PARALLEL POLL CONFIGURE command followed by PARALLEL POLL DISABLE. All devices that are currently addressed as listeners are disabled.

In the previous example a device at bus address 5 was configured to respond to parallel polls on D1. To disable the same device from responding you can use:

```
...
command[0] = MTA;          /*talk address of interface*/
command[1] = 63;          /* the UNLISTEN command*/
command[2] = 32 + 5;      /* the listen address for device at*/
                          /* address 5 */
command[3] = 5;          /* the PARALLEL POLL CONFIGURE command*/
command[4] = 112;        /* the PARALLEL POLL DISABLE command*/
hpib_send_cmd( eid, command, 5);
...
```

Conducting a Parallel Poll

Once the parallel poll responses of devices on the HP-IB have been configured (either remotely or locally), the Active Controller can conduct a parallel poll with *hpib_ppoll*.

The *hpib_ppoll* routine returns an integer whose least significant byte contains the 8-bit response to the parallel poll. Each device that is enabled to respond to a parallel poll places its status bit on a previously configured line. If an error occurs while the poll is being taken, a -1 is returned by the routine.

hpib_ppoll is invoked as follows:

```
hpib_ppoll( eid);
```

where *eid* is the entity identifier for the open interface file connected to the bus.

The code below indicates how you can interpret the byte returned by *hpib_ppoll*. Assume that a device at address 6 was previously configured to respond to a parallel poll by placing a 1 on D0 if it needed service. Assume the device at address 7 was configured to respond similarly on D1. If these are the only two devices able to respond to a parallel poll, you only care about the values of the 2 least significant bits of the integer returned by *hpib_ppoll*. The actual service routines have been left out of the example.

```
#include <fcntl.h>
main()
{
    int eid, status, byte;
    eid = open("/dev/raw_hpib", O_RDWR);

    if ((status = hpib_ppoll( eid)) == -1) /*conduct the parallel poll*/
    {
        printf("error taking ppoll"); /*if -1 returned then error occurred*/
        exit(1);
    }
    byte = status & 3;                /*set all but the least significant*/
                                      /*2 bits to zero          */
    switch (byte) {
        case 0:                        /*neither device is requesting service*/
            ...
            break;
        case 1:                        /*device at address 6 wants service*/
            ...
            break;
        case 2:                        /*device at address 7 wants service*/
            ...
            break;
        case 3:                        /*both devices want service*/
            ...
            break;
    }
}
```

Errors During Parallel Polling

The *hpib_ppoll* routine returns a -1 if any one of the following error conditions are true:

- The timeout defined by *io_timeout_ctl* occurred before all of the devices responded.
- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to a raw HP-IB interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value	Error Condition
-------------	-----------------

EBADF	<i>eid</i> did not refer to an open file.
-------	---

ENOTTY	<i>eid</i> did not refer to a raw interface file.
--------	---

EIO	The interface was not the Active Controller or a timeout occurred.
-----	--

Waiting For a Parallel Poll Response

The *hpib_wait_on_ppoll* routine allows you to wait for a specific parallel poll response from one or more devices. The effect of this is similar to waiting for the assertion of the SRQ line with *hpib_status_wait* (see the section *Servicing Requests*, presented earlier). With *hpib_wait_on_ppoll* you can wait for a specific device to request service; while *hpib_status_wait* is interrupted when any device requests service.

hpib_wait_on_ppoll is called with the form:

```
hpib_wait_on_ppoll( eid, mask, sense);
```

where *eid* is the entity identifier for the open interface file, *mask* is an integer whose binary value indicates on which parallel poll lines you are waiting for a request, and *sense* is an integer whose binary value indicates on which of these lines the request will use negative logic (device responds with 0 when it wants service). The routine returns the response byte **XOR**-ed with the *sense* value and **AND**-ed with the *mask*, unless an error occurs, in which case it returns a -1.

Calculating the mask

The routine *hpib_wait_on_ppoll* only looks at the least significant byte of the *mask* integer; therefore, the integer's remaining bytes can contain anything. For simplicity, the examples in this discussion set the upper bytes to zeros.

The *mask* value is determined as follows.

1. Decide which of the parallel poll lines (the 8 data lines) you want to wait for a request for service on. Assume that the lines are labeled D0-D7.
2. Set up an 8-bit binary number where the bits associated with the lines whose assertion you want to wait for are set to 1 and all of the other bits are 0. (D0 is associated with the least significant bit of the binary number, and D7 with the most significant.)
3. Given the binary number from step 2, calculate its decimal value. This is the *mask* integer you should use with *hpib_wait_on_ppoll*.

For example, assume that you want to wait for device A and device B to request service. You know that device A has been configured to respond on the parallel poll line D0 and device B has been configured to respond on line D4. The binary value of the *mask* that you will use is:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	0	0	0	1

The decimal value of this number is 17; the *mask* that you will use is 17.

Now consider a *mask* of 0. It indicates that you do not want to wait for a request on any of the parallel poll lines, meaning that a call to *hpib_wait_on_ppoll* using a *mask* of 0 has no effect.

Calculating the sense

The routine *hpib_wait_on_ppoll* also only looks at the least significant byte of the *sense* integer. For simplicity, the examples in this discussion set the upper bytes to zeros.

The *sense* value is determined as follows.

1. Decide which of the parallel poll lines (the 8 data lines) you want to wait for a request for service on. Assume that the lines are labeled D0-D7.
2. Determine which of these lines will indicate a request for service with a 0. This means that you must know the *sense* with which the associated devices are configured to respond to parallel polls.
3. Set up an 8-bit binary number where the bits associated with the lines that use a 0 to indicate a service request are set to 1 and all of the other bits are 0. (D0 is associated with the least significant bit of the binary number, and D7 with the most significant.)
4. Given the binary number from step 3, calculate its decimal value. This is the *sense* integer you should use with *hpib_wait_on_ppoll*.

Refer back to the example given for calculating the *mask* value. You know that device A is configured to respond on line D0 with a 1 when it wants service, but device B is going to request service with a 0 on line D4. The binary value of the *sense* that you will use is:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	0	0	0	0

The decimal value of this number is 16; the *sense* that you will use is 16.

If all of the devices on the bus respond to parallel polls with a 1 to request service, then the *sense* value can always be 0, no matter which parallel poll lines you are waiting for. If, on the other hand, all of the devices request service with a 0, then the *sense* value can always be 255 (11111111 in binary). You need only calculate a different *sense* value if devices on the bus respond with different levels.

An Example

Assume that you want to use *hpib_wait_on_ppoll* to wait until all of the devices on a bus are requesting service so that you can service them all at once. Your bus is configured as follows:

Device	Bus Address	Parallel Poll Response Line	Requests Service With A:
A	5	D0	1
B	7	D1	0
C	9	D2	0
D	11	D3	1

Begin by calculating the mask value for *hpib_wait_on_ppoll*. You want to wait for responses on lines D0, D1, D2, and D3; therefore, the *mask* value is:

Binary:	Decimal:
0 0 0 0 1 1 1 1	15

Since the four devices on the bus use both positive and negative logic, you must calculate the *sense* value. The devices responding on lines D1 and D2 use 0 to request service; therefore, the *sense* value is:

Binary:	Decimal:
0 0 0 0 0 1 1 0	15

Now that you have the *mask* and *sense* values you can write the code that makes the call to *hpib_wait_on_ppoll*:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open("/dev/raw_hpib", O_RDWR);

    if (hpib_wait_on_ppoll( eid, 15, 6) == -1)
        printf("either a timeout or error occurred");
    else
        service_routine();
}
```

In the code above, for *service_routine* to be executed all 4 of the devices must be requesting service with their parallel poll response. *Service_routine* should contain code that services all of the devices, either individually or as a group. See the hardware-dependency Appendix for any restrictions that may apply to your system.

Serial Polling

A sequential poll of individual devices on the bus is known as a **serial poll**. One entire byte of status is returned by the specified device in response to a serial poll. This byte is called the **status byte message** and, depending on the device, may indicate an overload, a request for service, or a printer being out of paper. The particular response of each device depends on the device.

Not all devices can respond to a serial poll. To find out if a particular device can be serially polled, consult its documentation. Trying to serially poll a device that cannot respond causes a timeout or suspends your program indefinitely.

The Active Controller cannot serial poll itself.

Unlike the parallel poll responses, serial poll responses cannot be configured remotely by the Active Controller. They are device-dependent and you must refer to a device's documentation to see how it responds.

Conducting a Serial Poll

The *hpib_poll* routine performs a serial poll of a specified device. It is called with the form:

```
hpib_poll( eid, address);
```

where *eid* is the entity identifier for the open interface file and *address* is the bus address of the device to be polled. The routine returns an integer, the lowest byte of which contains the status byte message (the serial poll response) from the addressed device. Only one device can be polled per call to *hpib_poll*.

Although the status byte message supplied by the addressed device is device-dependent, one bit always supplies the same information. Given that the status byte's bits are labelled D0-D7, D6 always indicates whether or not the device is requesting service by asserting the SRQ line.

The code below illustrates how *hpib_poll* can be used to find out if a device at bus address 5 is requesting service. It does this by asserting SRQ (it only looks at D6).

```
#include <fcntl.h>
main()
{
    int eid, status;
    eid = open("/dev/raw_hpib", O_RDWR);

    if ((status = hpib_poll( eid, 5)) == -1)    /*conduct serial poll*/
    {
        printf("error during serial poll");
        exit(1);
    }
    if (status & 64)                            /*after setting every bit except D6*/
                                                /*to zero; if D6 is set the device*/
                                                /*is requesting service */
        service_routine();
}
```

Errors During Serial Poll

The *hpib_spoll* routine returns a `-1` indicating an error if any of the following conditions are true:

- The addressed device did not respond to the serial poll before the timeout defined by *io_timeout_ctl* occurred.
- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value Error Condition

EBADF *eid* did not refer to an open file.

ENOTTY *eid* did not refer to a raw interface file.

EIO The device polled did not respond before the timeout or the interface was not the Active Controller.

Passing Control

The current Active Controller can pass the active control capability to a **Non-Active Controller** with the *hpib_pass_ctl* routine. A **Non-Active Controller** is a device capable of becoming Active Controller, and in most cases this means it is a computer.

hpib_pass_ctl is called as follows:

```
hpib_pass_ctl( eid, address);
```

where *eid* is the entity identifier for the open interface file (that is currently the Active Controller) and *address* is the bus address of a Non-Active Controller. Once the call is completed, the Non-Active Controller is the new Active Controller and the interface is a Non-Active Controller.

The *hpib_pass_ctl* routine only passes active control capability, it does not pass system control capability.

What If Control Is Not Accepted?

Your program is not suspended if the Non-Active Controller that you address does not accept active control of the bus. However, the computer still loses active control. This means the bus no longer has an Active Controller. If this happens, the System Controller must assume the role of Active Controller with *hpib_abort* (see *The System Controller's Duties* section) or *io_reset*.

No error is returned by *hpib_pass_ctl* if the device that you address does not accept active control. There is also no direct way to determine in advance if a given device can accept active control. However, if the computer immediately requests service after passing control and a timeout occurs before the request is acknowledged, possibly the active control wasn't accepted. There is no way for the computer, after initiating *hpib_pass_ctl*, to see if active control is accepted.

Errors While Passing Control

The routine `hpib_pass_ctl` returns a `-1` indicating an error if any of the following error conditions are true:

- The computer's interface is not the Active Controller.
- The `eid` entity identifier does not refer to an HP-IB raw interface file.
- The `eid` entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of `errno`:

Errno Value Error Condition

EBADF	<code>eid</code> did not refer to an open file.
ENOTTY	<code>eid</code> did not refer to a raw interface file.
EIO	The interface was not the Active Controller.

The System Controller

When the HP-IB's System Controller is first powered on or is reset, it assumes the role of Active Controller. An HP-IB can have only one System Controller. The System Controller cannot pass system control to any other controller (computer) on the bus. However, it can pass active control to another controller.

Determining System Controller

To find out if your computer's HP-IB interface is the System Controller, use the `hpib_bus_status` routine. It must be called as follows:

```
hpib_bus_status( eid, 3);
```

where `eid` is the entity identifier for the open interface file and `3` indicates that you want to find out if it is the System Controller. The routine returns a `1` if it is the System Controller, a `0` if it isn't, and a `-1` if an error occurs.

The code that follows prints a message indicating if the interface is the System Controller:

```
#include <fcntl.h>
main()
{
    int eid, status;
    eid = open("/dev/raw_hpib", O_RDWR);

    if ((status = hpib_bus_status( eid, 3)) == -1)
        printf("Error occurred during bus status routine");
    else if (status == 1)
        printf("Interface is the System Controller");
    else
        printf("Interface is not the System Controller");
}
```

System Controller's Duties

The System Controller of an HP-IB bus has three major functions:

- It assumes the role of Active Controller whenever it is powered on or reset.
- It can cancel talkers and listeners from the bus and assume the role of Active Controller by executing *hpib_abort*.
- It can control the logic level of the remote enable line (REN) with *hpib_ren_ctl*.

hpib_abort

A call to *hpib_abort* carries out the following actions:

- It terminates activity on the bus by pulsing the Interface Clear line (IFC). This results in all talkers and listeners on the bus being unaddressed.
- It sets the REN line so that devices on the bus will be placed in their remote state when they are addressed.
- It clears the ATN line if it was left set by the previous Active Controller.
- The System Controller then becomes the bus's new Active Controller.

The routine leaves the SRQ line unchanged, which means any device requesting service before *hpib_abort* is executed is still requesting service when the routine is finished.

To use *hpib_abort* on a particular HP-IB, the computer must be the System Controller of that bus. It does not have to be the Active Controller.

One situation where *hpib_abort* is useful is when the bus's Active Controller attempts to pass active control to another device that does not accept active control. This happens if the device addressed to receive control is not another controller. As a result the bus is left without any Active Controller and the System Controller must assume that role using *hpib_abort*.

hpib_abort is called as follows:

```
hpib_abort(eid);
```

where *eid* is the entity identifier for the open interface file.

hpib_ren_ctl

With *hpib_ren_ctl* you can enable or disable the REN line on the HP-IB. If the line is enabled, all devices that are capable of remote operation (interpreting HP-IB commands) can be placed in the remote state by the Active Controller addressing them as talkers or listeners. When REN is disabled, all devices on the bus return to their local state and cannot be accessed remotely.

When the System Controller is powered on or reset, the REN line is enabled by default. It is also enabled if the System Controller executes *hpib_abort*.

To use *hpib_ren_ctl* on a particular HP-IB , the computer must be the System Controller of that bus. It does not have to be the Active Controller.

hpib_ren_ctl is called as follows:

```
hpib_ren_ctl( eid, flag);
```

where *eid* is the file descriptor for the open interface file and *flag* is an integer. If *flag* is zero, the REN line is disabled. If it has any other value, then REN is enabled.

Errors During *hpib_abort* and *hpib_ren_ctl*

hpib_abort and *hpib_ren_ctl* both return a -1 indicating an error if any of the following error conditions are true:

- The computer's interface is not the System Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value Error Condition

EBADF	<i>eid</i> did not refer to an open file.
ENOTTY	<i>eid</i> did not refer to a raw interface file.
EIO	The interface was not the System Controller.

The Computer As a Non-Active Controller

Determining the Controller's Status

The *hpib_bus_status* routine allows you to determine information about the interface card and the HP-IB. It can be used by any controller on the bus, independent of whether or not it is the Active Controller or System Controller. In the previous discussions about the Active and System Controllers, the routine is mentioned briefly. The discussion that follows should give you a broader look at the routine's uses.

hpib_bus_status is called with the form:

```
hpib_bus_status( eid, status_question );
```

where *eid* is the entity identifier for the open interface file and *status_question* is an integer that indicates what question you want answered. The value of *status_question* must be within the range 0-7 where the value indicates the following questions:

Value	Status Question
0	Is the interface in the remote state?
1	Are there any devices requesting service? (Is SRQ asserted?)
2	Is there a listener that is not ready for data? (Is NDAC asserted?)
3	Is the interface the System Controller?
4	Is the interface the Active Controller?
5	Is the interface currently addressed as a talker?
6	Is the interface currently addressed as a listener?
7	What is the interface's bus address?

If the value of *status_question* is in the range 0-6, the routine returns a 1 if the answer to the question is yes or a 0 if the answer is no. If the value of *status_question* is 7, the routine returns the bus address of the computer's interface. If *status_question* has any other value, a -1 is returned, indicating an error.

For example, to determine if your interface is a Non-Active Controller on the bus, use the routine call illustrated by the following code:

```
...
...
if ((status = hpib_bus_status( eid, 4)) == -1)
    printf("Error occurred while checking status");
else if (status == 0)
    printf("Computer is a Non-Active Controller");
else
    printf("Computer is the Active Controller");
...
...
```

Requesting Service

When your computer is a Non-Active Controller it can request service from the current Active Controller by asserting the SRQ line. This is done with the *hpib_rqst_srvce* routine. The routine is called as follows:

```
hpib_rqst_srvce( eid, response);
```

where *eid* is the entity identifier for the open interface file and the lowest byte of *response* is the integer value of the 8-bit response that the computer gives if it is serially polled. The upper bytes of *response* are ignored by the routine. Given a bit labeling of D0-D7, D6 of the lower byte sets the SRQ line. The defined values for the remaining 7 bits are device-dependent. This section only discusses the setting and clearing of the SRQ line with D6 (integer value of 64).

To request service you can invoke *hpib_rqst_srvce* as follows:

```
#include <fcntl.h>
main()
{
    int eid;

    eid = open("/dev/raw_hpib", O_RDWR);
    hpib_rqst_srvce( eid, 64); /*Bit 6 of serial poll response is set*/
                               /*and SRQ is asserted                */
}
```

Note that by setting *response* to 64 the only information that the Active Controller receives when it serially polls your computer is that you are asserting the SRQ line.

hpib_rqst_srvce returns a 0 if it executes correctly or a -1 if an error occurred.

Once you have asserted SRQ, the line remains asserted until the Active Controller serially polls you or you call *hpib_rqst_srvce* again and clear bit 6 (e.g. *hpib_rqst_srvce(eid, 0)*). After your serial poll response is configured, your computer's interface responds automatically to any serial polls from the Active Controller.

Note that if another device is asserting SRQ also, the line is still asserted after your request is removed.

If you try to request service and you are the Active Controller, the SRQ line is not set. However, if you then pass active control to another computer, the *response* that you specified with *hpib_rqst_srvce* is remembered and the SRQ line is set.

When the Active Controller sees the SRQ line asserted, it usually polls the devices on the bus to find out who is requesting service. To determine which device (or devices) is requesting service, the Active Controller conducts a parallel poll. Configuring your computer's response to a parallel poll is discussed in the next section.

If a device responds to a parallel poll with an **I need service** message, the Active Controller can perform a serial poll to determine what service action is required. If several devices are configured to respond to a parallel poll on the same line and the Active Controller sees that line is requesting service, it must perform a serial poll of each of the devices to find out which one is requesting service.

Errors While Requesting Service

The routine *hpib_rqst_srvice* returns a `-1` indicating an error if either of the following error conditions are true:

- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value Error Condition

EBADF *eid* did not refer to an open file.

ENOTTY *eid* did not refer to a raw interface file.

Responding to Parallel Polls

Before your computer can respond to a parallel poll from the Active Controller, its response must be configured. This can be programmed remotely by the Active Controller (see *The Active Controller* section) or locally using *hpib_card_ppoll_resp*.

Configuring a parallel poll response of a device involves:

- Specifying the logic sense of the response (i.e. whether a 1 means the device does or doesn't need service).
- Specifying which data line the device responds on. More than one device can be configured to respond on the same line.

To locally configure how your computer responds to parallel polls, call *hpib_card_ppoll_resp* as follows:

```
hpib_card_ppoll_resp( eid, flag);
```

where *eid* is the entity identifier of the open interface file and *flag* is an integer whose binary value configures the response.

Calculating the Flag

The *flag* value is found by first forming an 8-bit binary number and then using the decimal value of that number. The binary number's bits have the following meaning:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	S	P	P	P

where:

- S** sets the sense of the response if allowed by the hardware. If *S* is a 1, then the device responds with a 1 when it requires service.
- P** is a 3-bit binary number that specifies which of parallel poll response lines (D0-D7) the device responds on if allowed by the hardware.

Limitations of `hpib_card_ppoll_resp`

There are some hardware limitations on using `hpib_card_ppoll_resp` to configure parallel poll responses. You should refer to the Appendix for your system to find out if any restrictions apply. If there are restrictions on your system, you may find it easier to configure the interface's parallel poll response remotely with the Active Controller. Note the Active Controller can configure its own response, but the response only has effect when it passes active control.

Errors While Configuring Response

The routine `hpib_card_ppoll_resp` returns a `-1` indicating an error if any of the following error conditions are true:

- The interface cannot respond on the line number specified by *flag*.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value Error Condition

- `EBADF` *eid* did not refer to an open file.
- `ENOTTY` *eid* did not refer to a raw interface file.
- `EINVAL` The interface cannot respond on the line indicated by *flag*.

Hpip_ppoll_resp_ctl

The *hpip_ppoll_resp_ctl* function allows the user to determine how the computer will respond to the next parallel poll. There are two ways to respond to a parallel poll. Responding favorably indicates to the controller that the computer wants to be serviced. Responding unfavorably indicates the computer does not need the Active Controller's attention.

The parallel poll response is set as follows:

```
hpib_ppoll_resp_ctl(eid, response_value);
```

where *eid* is the entity identifier of an open interface file and the *response_value* is an integer that indicates the response to use. If *response_value* is non-zero then the computer will respond favorably to the next parallel poll. A zero *response_value* will respond unfavorably to the next parallel poll.

Disabling Parallel Poll Response

The function *hpib_card_ppoll_resp* also allows you to disable your interface from responding to parallel polls made by the Active Controller. This is done by setting bit D4 of the routine's flag value. When D4 is 0 the interface is enabled to respond to parallel polls, and when it is 1 the interface's parallel poll response is disabled. Thus, a flag value of 16 disables the response. For example, the code:

```
...
...
hpib_card_ppoll_resp( eid, 16); /*disable parallel poll response*/
...
...
```

disables the interface with the entity identifier *eid* from responding to any parallel polls.

Accepting Active Control

If your computer is a Non-Active Controller, the current Active Controller may pass active control to you. Your computer's interface accepts active control automatically; however, you must design an interfacing program to recognize when this happens.

The *hpib_bus_status*, *hpib_status_wait*, and *io_on_interrupt* routines allow recognizing the computer has become the Active Controller.

hpib_status_wait has been mentioned in previous discussions about the Active Controller and System Controller. The following discussion provides a look at its uses.

hpib_status_wait is called as follows:

```
hpib_status_wait( eid, status);
```

where *eid* is the entity identifier for the open interface file and *status* is an integer indicating what condition you want to wait for. The following values for *status* are defined:

Value	Condition Waiting For
1	Wait until the SRQ line is asserted
4	Wait until this computer is the Active Controller
5	Wait until this computer is addressed as a talker
6	Wait until this computer is addressed as a listener

Now imagine a situation where the current Active Controller is programmed to know that when your computer requests service it is to pass active control to you. The following code shows how you can program your computer to request service and then wait until it becomes the bus's new Active Controller.

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open("/dev/raw_hpib", O_RDWR);
    if (hpib_rqst_srvce( eid, 64) == -1) /*set SRQ line to request service*/
    {
        printf("Error while requesting service");
        exit(1);
    }

    if (hpib_status_wait( eid, 4) == -1) /*wait until Active Controller*/
    {
        printf("Error while waiting for status");
        exit(1);
    }
    ... /*Computer is now the Active Controller*/
    ...
    ...
}
```

Notice for *hpib_status_wait* to have returned a -1 (due to a timeout occurring, you would have had to set a timeout value, using *io_timeout_ctl*, after opening the interface file. Since this wasn't done in the example above, no timeout occurs.

Errors While Waiting on Status

hpib_status_wait returns a -1 indicating an error if any of the following error conditions are true:

- A timeout occurred before the condition the routine was waiting for became true.
- The *status* specified has an undefined value.
- The *eid* entity identifier does not refer to a raw HP-IB interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, your program should check for the following values of *errno*:

Errno Value Error Condition

EBADF	<i>eid</i> did not refer to an open file.
ENOTTY	<i>eid</i> did not refer to a raw interface file.
EINVAL	<i>status</i> contains an invalid value.
EIO	The specified condition did not become true before a timeout occurred.

Determining When You Are Addressed

As a Non-Active Controller you may be addressed by the Active Controller and become a bus talker or listener for data transfer. The DIL routines *hpib_bus_status*, *hpib_status_wait*, and *io_on_interrupt* allow you to find out if the computer's interface is currently being addressed.

The following code determines if the interface is currently addressed as a bus talker:

```
#include <fcntl.h>
main()
{
    int eid;
    eid = open("/dev/raw_hpib", O_RDWR);
    if (hpib_bus_status( eid, 5) == 1)
    {
        printf("the interface is addressed as a talker");
        write( eid, "data message", 12);  /*do the expected data transfer*/
    }
    else
        printf("the interface is not addressed as a talker");
}
```

In the above call to *hpib_bus_status*, *eid* is the entity identifier for the interface and 5 indicates that you are asking if it is a bus talker. The routine returns a 1 if the answer is yes and 0 if the answer is no.

To find out if the interface is currently addressed as a bus listener use the following:

```
...
...
if (hpib_bus_status( eid, 6) == 1)
{
    printf("the interface is addressed as a listener");
    read( eid, buffer, 12);          /*do the data transfer*/
}
else
    printf("the interface is not addressed as a listener");
...
...
```

If you need to wait until the interface is addressed as either a talker or listener and then handle a data transfer, use the DIL routine *hpib_status_wait*. When you call the routine, you specify the entity identifier of the interface and the bus condition that you want to wait on:

```
hpib_status_wait( eid, condition);
```

As with *hpib_bus_status*, with a condition value of 5 the routine waits for the interface to be addressed as a talker. With a condition value of 6 the routine waits until it is a listener. How long the routine waits for the specified condition is controlled by the timeout value that you have previously set for the entity identifier with *io_timeout_ctl* (see discussion in *Chapter 2: General Purpose Routines*). The routine returns a 0 if the condition became true or a -1 if a timeout (or an error) occurred first.

In the example below, the program waits for the interface to become a bus listener and then it reads a 50-byte message.

```
#include <fcntl.h>
main()
{
    int eid;
    char buffer[50];          /*storage for message*/
    eid = open("/dev/raw_hpib", O_RDWR);
    io_timeout_ctl( eid, 500);

    if (hpib_status_wait( eid, 6) == -1)
    {
        printf("Either a timeout or an error occurred");
        exit(1);
    }

    read( eid, buffer, 50);   /*read data into buffer*/
    printf("Message is: %s", buffer); /*print data message*/
}
```

Note that a timeout value is set for the interface file's entity identifier in the code above so the program does not hang while waiting for the interface to be addressed as a bus listener.

The following example illustrates how to use the *io_on_interrupt* routine to set up an interrupt handler to handle a data transfer:

```
#include <dvio.h>
#include <fcntl.h>
int eid;
char buffer[50];
main()
{
    int eid;
    struct interrupt_struct cause_vec;
    eid = open("/dev/raw_hpib", O_RDWR);
    cause_vec.cause = TLK;
    io_on_interrupt(eid, cause_vec, handler);
    ...
    ...
}
handler(eid, cause_vec);
int eid;
struct interrupt_struct cause_vec;
{
    if (cause_vec.cause == TLK)
        read(eid, data, 50);
}
```

Buffering I/O Operations

The DIL routine *hpib_io* allows you to perform structures of HP-IB I/O operations for both sending HP-IB commands and transferring data. The computer's HP-IB interface must be the bus's Active Controller before this routine can be used.

A call to *hpib_io* has the form:

```
#include <dvio.h>
int eid;
struct iodetail *iovec;
int iolen;
...
...
hpib_io( eid, iovec, iolen);
```

where *eid* is the entity identifier of the open interface file, *iovec* is a pointer to an array of I/O operation structures, and *iolen* is the number of structures in the array. The name of the template for the I/O operation structures is *iodetail* and it is defined in the include file *dvio.h*.

Iodetail: The I/O Operation Template

The form of the *iodetail* structure that holds I/O operations is:

```
struct iodetail {
    char mode;
    char terminator;
    int count;
    char *buf;
};
```

Each of the components of *iodetail* have the following meanings:

- mode* Describes what kind of I/O operation the structure contains.
- terminator* Specifies whether or not there is a read termination character for the I/O operation, and if so it specifies the value.
- count* How many bytes are to be transferred during the I/O operation.
- buf* A pointer to an array containing the bytes of data to be transferred.

Components of a particular *iodetail* structure are referenced with:

```
iovec->component
```

where *iovec* is a pointer to an array of *iodetail* structures and *component* is either *mode*, *terminator*, *count*, or *buf*.

The Mode Component

The *mode* describes what type of I/O is to be performed on the data pointed to by the *buf* component. You determine its value by **OR**-ing constants from a set defined in the include file *dvio.h*. The constants that you can choose from are:

Name	Description
HPIBREAD	Perform a read operation and place the data into the accompanying buffer pointed to by <i>buf</i> . Can be by itself or OR -ed with HPIBCHAR.
HPIBWRITE	Perform a write operation using the data in the accompanying buffer pointed to by <i>buf</i> . Can be by itself or OR -ed with either HPIBATN or HPIBEOI but not both.
HPIBATN	If you are performing a write operation, the data is placed on the bus with ATN asserted (you are sending a bus command). It only has effect if you also specify HPIBWRITE.
HPIBEOI	If you are performing a write operation, the EOI line is asserted when the last byte of data is sent. It only has effect if you also specify HPIBWRITE.
HPIBCHAR	If you are performing a read operation, the transfer is halted when the <i>terminator</i> component value of the <i>iodetail</i> structure is read. The <i>terminator</i> component only has effect if you OR HPIBCHAR and HPIBREAD. The HPIBCHAR constant only has effect if also specify HPIBREAD.

NOTE

When you construct *mode*, you must use either HPIBREAD or HPIBWRITE, but not both. Optionally, you can **OR** one of the other three constants with either HPIBREAD or HPIBWRITE, but they are not required. HPIBCHAR only has effect when it is **OR**-ed with HPIBREAD, while HPIBATN and HPIBEOI only have effect when they are **OR**-ed with HPIBWRITE (but not both at the same time).

The *mode* component allows you to specify under what conditions an I/O operation terminates. All I/O operations terminate when the maximum number of bytes specified by the *count* component of the *iodetail* structure is reached. However, additional termination conditions are possible:

- If you specify HPIBREAD and HPIBCHAR, the detection of the termination character determined by the *terminator* component also causes termination.
- If you specify HPIBWRITE and HPIBEOI, when the count value is reached EOI is asserted at the time that the last byte of data is sent (unless you also specify HPIBATN).

To illustrate, assume that *iovec* points to an *iodetail* structure that you are building and you want the structure to send several HP-IB commands. The *mode* component of the structure is assigned the necessary value as follows:

```
iovec->mode = HPIBWRITE | HPIBATN;
```

The Terminator Component

The *terminator* component of the *iodetail* structure specifies a character that causes the termination of a read operation when it is detected. The *terminator* only has effect if HPIBREAD/HPIBCHAR is specified as the structure's associated *mode* component.

Assign a value to the *terminator* of the structure pointed to by *iovec* with:

```
iovec->terminator = value;
```

For example, to make the ASCII period (".") the termination character, use the statement:

```
iovec->terminator = '.';
```

The Count Component

The *count* is an integer determining the maximum number of bytes that will be transferred during the structure's I/O operation. Reading or writing always terminates when this value is reached, but additional termination conditions can be set up using the structure's associated *mode* component.

Set a maximum number of bytes for a structure's data transfer with:

```
iovec->count = max_value;
```

where *iovec* is a pointer to the structure and *max_value* is an integer.

The Buf Component

The *buf* component points to a character array that holds the data that will be transferred during a read operation (HPIBREAD) or is written to during a write operation (HPIBWRITE). Note the array's size limit is defined by the structure's *count* component.

One way to store a message in the *buf* array is:

```
iovec->buf = "data message";
```

Allocating Space

Before you can build *iodetail* structures for your I/O operations, you need to allocate space for them in memory. The easiest way to do this (if you are programming in C) is to write a routine that allocates space for *n* *iodetail* structures and returns a pointer to the first one.

Below is the code for such a routine, *io_alloc*:

```
struct iodetail *io_alloc(n);
int n;
{
    char *malloc();
    return((struct iodetail *) malloc(sizeof(struct iodetail) * n));
}
```

Refer to the *HP-UX Reference* for a description of *malloc(3C)*.

To use *io_alloc* to allocate memory space for 10 *iodetail* structures your program should contain the statements:

```
struct iodetail *iovec;    /*define an iodetail pointer*/
iovec = io_alloc(10);      /*allocate space for 10 iodetail structures*/
```

An Example

Assume that your computer's HP-IB interface is at HP-IB address 3 and it is the bus's Active Controller. You want to send a data message to a device at HP-IB address 7 and then receive a message from the same device using *hpib_io*. Four *iodetail* structures are required to do this:

1. The first structure configures the bus so that the interface is the talker and the device at address 7 is the listener.
2. The second structure sends the data message from the interface to the device.
3. The third structure configures the bus so that the device at address 7 is the talker and the interface is the listener.
4. The fourth structure receives the data message from the device.

The code below illustrates how the 4 structures can be built and then implemented.

```
#include <fcntl.h>
#include <dvio.h>          /*contains definitions for iodetail*/
struct iodetail *io_alloc(n);
int n;
{
    char *malloc();
    return ((struct iodetail *) malloc(sizeof (struct iodetail) *n));
}
main()
{
    extern int errno;
    int eid;
    char buffer[4][12];
    struct iodetail *iovec, *temp; /*2 pointers to iodetail structures*/

    /*Allocate space for 4 iodetail structures*/
    eid = open("/dev/raw_hpib", O_RDWR);
    iovec = io_alloc(4);          /*use the routine described earlier*/
    temp = iovec;

    /*Build structure 1 -- Configuring the bus*/
    temp->mode = HPIBWRITE | HPIBATN; /*you want to send commands*/
    strcpy(buffer[0], "?^ "); /*address computer to talk and bus address to listen*/
    temp->buf = buffer[0];
    temp->count = strlen(temp->buf);

    /*Build structure 2 -- Sending the data message*/
    temp++; /*use temp pointer so that iovec remains pointing to the*/
           /*first structure but temp now points to the next one*/

    temp->mode = HPIBWRITE | HPIBEOI; /*you want EOI asserted when the
                                     transfer is done*/

    strcpy(buffer[1], "data message");
    temp->buf = buffer[1];
    temp->count = strlen(temp->buf);

    /*Build structure 3 -- Configuring the bus*/
    temp++; /*increment structure pointer*/
    temp->mode = HPIBWRITE | HPIBATN; /*you want to send commands*/
    strcpy(buffer[2], "?@>");
    temp->buf = buffer[2];
    temp->count = strlen(temp->buf);

    /*Build structure 4 -- Receiving data message*/
    temp++; /*increment structure pointer*/
    temp->mode = HPIBREAD /*read data; reaching count value terminates read*/
    temp->count = 10; /*you expect a 10-byte message*/
    temp->buf = buffer[3];
}
```

```

/*Implement the I/O operations stored in the iodetail structures*/
    eid = open("/dev/raw_hpib", O_RDWR);
    hpib_io( eid, iovec, 4);

    if (hpib_io(eid, iovec, 4) == -1)
    {
        printf ("hpib_io failed\n");
        printf ("errno = %d\n",errno);
        exit(1);
    }

/*Print data message you received from the device. Note temp still*/
/*points to the last iodetail structure and the last structure did the read*/
    printf("%s", temp->buf);
}

```

One comment about the C language; routine parameters are passed by value and not by reference; therefore, after you execute *hpib_io* the *iovec* parameter still points to the first *iodetail* structure, just as it did before the routine executed. Thus, another way to print out the data message, read into the *buf* component of the fourth *iodetail* structure in the example above, is:

```
printf("%s", (iovec[3]->buf));
```

Locating Errors in Buffered I/O Operations

If all of the I/O operations specified in the array of *iodetail* structures complete successfully, *hpib_io* returns a 0 and updates the *count* component of each structure to reflect the actual number of bytes read or written.

If an error occurs during one of the I/O operations, *hpib_io* immediately returns a -1 indicating the error. To find out during which *iodetail* structure operation the error occurred, look at the structures' *count* components. The *hpib_io* routine updates the *count* component of the structure that caused the error to be a -1. Once you have located a structure with a count of -1, you know that all of the structures previous to it were completed successfully and all of the structures after it were not executed at all.

For example, assume that you have built an array of 10 *iodetail* structures to execute a sequence of I/O operations. The following code executes the operations and then checks for errors. If an error occurs, the code prints the number of the structure that caused it (for instance, the first structure in the array is number 1).

```
#include <fcntl.h>
#include <dvio.h>
main()
{
    int FOUND, number, eid;
    struct iodetail, *iovec, *temp;
    ...
    /*space is allocated for the 10 structures and then they are*/
    /*built. "Iovec" is left pointing to the first structure*/
    ...
    eid = open("/dev/raw_hpib", O_RDWR); /*open the interface file*/

    if (hpib_io( eid, iovec, 10) == -1) /*execute the operations and if a -1*/
        /*is returned then an error occurred*/
    {
        number = 1;           /*initialize counter*/
        FOUND = 0;           /*initialize Boolean flag*/
        temp = iovec;        /*set temporary pointer to first structure*/
        while (number <= 10 && FOUND != 1)
            if (temp->count == -1) /*found structure that caused error*/
                FOUND = 1;
            else
                {
                    temp++;           /*move pointer to next structure*/
                    number++;         /*increment counter*/
                }
        if (FOUND == 1)
            printf("Structure number %d caused error", number);
        else
            printf("Error but couldn't find structure that caused it");
    }
    else
        printf("No error occurred during execution of hpib_io");
}
```

Controlling the GPIO Interface

4

This chapter briefly describes the actions you take to configure your GPIO interface before it can be accessed from a program using the DIL routines. It then discusses the limitations and capabilities that DIL provides for controlling the GPIO interface.

Configuring Your GPIO Interface

Setting the Interface Switches

The GPIO interface card has several switches that allow you to configure your interface. These are fully described in the interface's installation manual. The functions they configure are:

- the data logic sense
- the data handshake mode
- the input data clock source
- whether or not the computer checks the Peripheral Status line (PSTS) before initiating a data transfer

Set the switches according to the directions found in the GPIO installation manual.

NOTE

On some systems, the GPIO interface's select code is determined by a switch setting on the interface card. Refer to the Appendix at the end of this article to see if a switch configuration is required. If a switch setting is not required, then the select code is determined by the I/O slot in which you place the interface card.

Creating the GPIO Interface File

Once you have set the necessary switches on your GPIO interface you must install the card in your computer and create an interface file for it. *Chapter 2: General Purpose Routines* discusses using *mknod* to create a special file for accessing the interface. You must create an interface file before you can access the interface from HP-UX. Refer to *Chapter 2: General Purpose Routines* for information on how to use *mknod* to create the interface file.

Limitations on Controlling the Interface

The Device I/O Library (DIL) routines allow you to use a GPIO interface to communicate with devices that are not supported on your HP-UX system. They do not provide you with full control of the interface and because of this, you are faced with the following limitations:

- You cannot recognize interrupts sent by the peripheral on the External Interrupt Request line (EIR).
- You do not have direct access to the interface's handshake lines: the Peripheral Control line (PCTL), the Peripheral Flag line (PFLG), and the Input/Output line (I/O).
- You cannot read the value of the Peripheral Status line (PSTS).

Using the DIL Routines

Several of the DIL routines can be used to control the GPIO interface. These are divided into two groups:

- general purpose routines used with either an HP-IB or GPIO interface
- GPIO routines; routines specifically designed to be used with a GPIO interface

The general purpose routines are listed and described in *Chapter 2: General Purpose Routines* and you should refer there for more information. They are used in this chapter to illustrate various aspects of controlling the GPIO interface from an HP-UX process.

There are two DIL routines that are restricted to use with a GPIO interface:

- *gpio_get_status*
- *gpio_set_ctl*

On the Series 500, these two routines allow you to use the four special purpose lines that are available on the interface for any purpose desired. The *gpio_get_status* routine reads the two lines controlled by the peripheral (STI0 and STI1) and *gpio_set_ctl* sets the values of the two lines controlled by the computer (CTL0 and CTL1). These two routines are described later in this chapter in the section *Using the Special Purpose Lines*.

By using the DIL general purpose routines and these two GPIO-specific routines you can:

- reset the interface
- perform data transfers
- use the interface's 4 special purpose lines
- control the data path width and data transfer speed
- set a timeout for data transfers
- set a read termination character
- get the termination reason
- set up the interrupts
- enable or disable interrupts

Resetting the Interface

The interface should always be reset before it is used, to ensure it is in a known state. All interfaces are automatically reset when your computer is powered on, but you can also reset them from your I/O process using the *io_reset* routine. For example, the following code resets a GPIO interface:

```
int eid;                               /*entity identifier*/
eid = open( "/dev/raw_gpio", O_RDWR); /*open GPIO interface file*/
io_reset(eid);                          /*reset the interface*/
```

This has the following effect:

- the Peripheral Reset line (PRESET) is pulsed low
- the PCTL line is placed in the clear state
- if the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logical 0)

The lines that are left unchanged are:

- the CTL0 and CTL1 output lines
- the I/O line
- the Data Out lines, if the DOUT CLEAR jumper is not installed

Performing Data Transfers

Using the DIL routines *read* and *write* you can transfer bytes of ASCII data to and from the GPIO interface. The following code illustrates using these routines to first write 16 bytes of data and then read 16 bytes.

```
int eid;                               /*entity identifier*/
char read_buffer[16], write_buffer[16]; /*buffers to hold data*/
eid = open( "/dev/raw_gpio", O_RDWR); /*open interface file*/
write_buffer = "message to write";     /*data message to send*/
write( eid,write_buffer, 16);          /*send message*/
read( eid, read_buffer, 16);           /*receive message*/
printf("%s", read_buffer);             /*print received message*/
```

Using the Special-Purpose Lines

On the Series 500, four special-purpose signal lines are available for a variety of uses. Two of the lines are for output (CTL0 and CTL1), and two are for input (STIO and STI1). The routine *gpio_set_ctl* allows you to control the values of CTL0 and CTL1, while the routine *gpio_get_status* allows you to read the values of STIO and STI1.

Driving CTL0 and CTL1

The call to *gpio_set_ctl* has the following form:

```
gpio_set_ctl(eid, value);
```

where *eid* is the entity identifier for the open GPIO interface file and *value* is an integer whose least significant two bits are mapped to CTL0 and CTL1.

To illustrate:

```
int eid;                /*entity identifier*/
eid = open("/dev/raw_gpio", O_RDWR); /*open interface file*/
gpio_set_ctl( eid, 3);   /*assert CTL0 and CTL1*/
```

Both CTL0 and CTL1 are asserted low; thus, in the above example both lines are pulled low. This logic polarity cannot be changed. To raise both of the lines, call *gpio_set_ctl* with:

```
gpio_set_ctl( eid, 0);
```

Reading STIO and STI1

The call to *gpio_get_status* has the following form:

```
int eid, value;
value = gpio_get_status(eid);
```

where *eid* is the entity identifier for the open GPIO interface file. The routine returns an integer whose least significant two bits are the values of STIO and STI1.

To illustrate:

```
int eid;                /*entity identifier*/
int value, bits;
eid = open("/dev/raw_gpio", O_RDWR); /*open interface file*/
value = gpio_get_status(eid);        /*look at STIO and STI1*/
bits = value & 03 /*clear all but the 2 least significant bits*/
if (bits == 3) /*and see if they're both set*/
    ...
    /*insert code that handles case when both STIO and STI1 are asserted*/
else if (bits == 1) /*just STIO is asserted*/
    ...
    /*insert code that handles case when STIO is asserted*/
else if (bits == 2) /*just STI1 is asserted*/
    ...
    /*insert code that handles case when STI1 is asserted*/
else /*neither are asserted*/
    ...
    /*insert code that handles case when neither STIO nor STI1 is asserted*/
```

Note that STIO and STI1 are asserted low; thus, when the value returned by *gpio_get_status* has one of its two least significant bits set, the associated special-purpose line is low.

Controlling the Data Path Width

The DIL routine *io_width_ctl* allows you to specify two different data path widths for your GPIO interface: 8 bits and 16 bits. The call has the following form:

```
io_width_ctl( eid, width);
```

where *eid* is the entity identifier for the open GPIO interface file and *width* is either 8 or 16. If a different *width* value is specified, the routine returns an error of -1 and *errno* is set to *EINVAL*. The GPIO interface defaults to an 8-bit path when its file is first opened.

The code below illustrates data transfers using a 16-bit data path.

```
int eid;
eid = open("/dev/raw_gpio", O_RDWR);          /*open the interface file*/
io_width_ctl( eid, 16);                       /*set path width at 16 bits*/
write( eid, "data message", 12);             /*perform data transfer*/
```

Since the interface's data path is 16 bits, 2 ASCII characters are transferred for each handshake cycle involved. In the first 16-bit transfer, *d* is sent in the upper byte and *a* is sent in the lower. The actual logic level of the GPIO data output lines depends on how the lines have been configured.

Controlling the Transfer Speed

You can request a minimum speed for the data transfer across a GPIO interface using *io_speed_ctl*. Your system rounds the speed that you specify up to the nearest defined speed. If you specify a speed that is faster than your system allows, the highest allowable speed is used. Refer to *Chapter 2: General Purpose Routines* for more information on using this routine. Again, the Series 500 always provides DMA; therefore, the routine *io_speed_ctl* is ineffective on that system.

In Case of a Timeout

If you have previously set a timeout value for the data transfer entity identifier, reaching the timeout after attempting a transfer will cause an error condition. If a timeout does occur, the DIL routine that you called to implement the transfer returns -1 and sets *errno* to *EIO*. When a timeout occurs you should reset the GPIO interface with the *io_reset* routine before attempting the transfer again.

Read Terminations

Determining Why a Read Operation Terminated

Chapter 2: General Purpose Routines describes a DIL routine called *io_get_term_reason* that is used to find out why the last read, done on a particular entity identifier, terminated. It tells you which of the following caused the termination:

- the requested number of bytes were read
- a specified read termination character was seen
- the assertion of the PSTS was seen
- some abnormal condition occurred, such as an I/O timeout

Specifying a Read Termination Character

Chapter 2: General Purpose Routines describes the routine *io_eol_ctl* which allows you to specify a character that when read will terminate the read operation on a particular entity identifier for the GPIO interface file.

Interrupts

Chapter 2: General Purpose Routines describes *io_on_interrupt* and *io_interrupt_ctl*. These routines allow you to set up and control interrupt handlers for the GPIO status line or a particular *eid* for the GPIO interface file.

Series 500 Dependencies

A

There are four areas of Series 500 system dependent information:

- information about creating the special file for the interfaces that you plan to access with DIL routines
- the relationship between entity identifiers and file descriptors
- the restrictions imposed by the hardware on using the DIL routines
- information about how you can improve the performance of your I/O process

Creating the Interface File

There are two areas of hardware-specific information that you must know before you can create a special file for an interface:

- the number of the driver that is required to communicate with the interface
- how the select code for the interface is determined

Determining the Driver

You specify the driver that is to be used with an interface in the **major number** argument of the *mknod* command. On the Series 500, the driver numbers that you use are:

Driver Number	Use
12	HP 27110A/B HP-IB Interface
18	HP 27110A GPIO Interface
37	Internal 550 HP-IB Interface

Determining the Select Code

You specify the select code for an interface as a two digit hexadecimal value component of the **minor number** argument of *mknod*. On the Series 500, the select code corresponds to the I/O slot in which the interface is placed.

Determining The Bus Address of the Interface Card

The HP 27110A/B card always assumes bus address 30 when it is the Active Controller. If control is passed, then it assumes the address specified by the cards switch setting. However, the *hpib_bus_status* routine always returns the correct bus address.

Entity Identifiers

On the Series 500, an entity identifier for a file used by a DIL routine is equivalent to an HP-UX file descriptor. This means that you can obtain entity identifiers for your interface files with the system routines *dup*, *fcntl*, and *pipe*, in addition to *open*.

Restrictions Using the DIL Routines

This section presents some restrictions on using the DIL routines on the Series 500 computers. These restrictions are organized under the routine to which they apply. The routines are presented in alphabetical order.

hpib_bus_status

A bug in the HP 27110A HP-IB interface card can cause an erroneous report of the state of the SRQ line. There is a small window when *hpib_bus_status(eid, 1)* reports that the line is clear when in reality it is set. Since the routine will never report that the line is set when in reality it is clear, **OR**-ing together successive readings of the state of the SRQ line minimizes the possibility of error. **OR**-ing five successive readings gives you a result that is approximately 99% accurate. This bug has been fixed in the HP 27110B card.

On the Series 500, it is possible to look at the SRQ line with *hpib_bus_status* and not see it asserted when it actually is. Because of this, you should check the SRQ line at least 5 times before determining whether or not it is asserted. If it is seen true any one of the 5 times, then the line is asserted (it will never be seen asserted when it actually isn't). For example:

```
#include <fcntl.h>
main()
{
    int eid, value, i;

    eid = open("/dev/raw_hpib", O_RDWR);
    value = 0;
    for ( i=0; i<5; ++i)
        value = hpib_bus_status( eid,1) + value;
    /*Notice that if SRQ is ever seen true, then "value" will be
    greater than 0*/

    if (value>0)
        service_routine();          /*SRQ is asserted; service the request*/
    else
        printf("No one is requesting service");
}
```

hpib_card_ppoll_resp

The HP 27110A/B HP-IB interface cards do not support programmatic configuration of their parallel poll response. The parallel poll response is set and enabled by the *hpib_card_ppoll_resp* routine. The default *sense* of the HP 27110A/B interface's parallel poll response is always 1. If the interface's address is 7 or less, the address determines the response's line number as follows: given that the bus data lines are labeled D0 through D7, they correspond to addresses 7 through 0, respectively. For instance, the parallel poll response of an HP 27110A/B with address 0 is a 1 on data line D7. If its address is 7 then it responds with a 1 on line D0. If the address of the interface is greater than 7, there is no default line for it to respond on. Therefore, unless its response is configured remotely by the Active Controller, it can not respond at all.

If you want the interface to respond with a sense of 0 or on a different line than HP 27110A/B defaults to, you must configure it remotely with the Active Controller

hpib_rqst_srvce

This routine provides the capability of configuring an HP-IB interface's 8-bit response to serial polls. However, the HP 27110A/B HP-IB interface only allows you to set bit 6 of the response; all the other bits are cleared. If you set bit 6 of the serial response (where the response bits are labeled bit D0-D7) and the interface is not the Active Controller, then the SRQ line is asserted. The line remains asserted until the interface is serially polled or you clear bit 6 with *hpib_rqst_srvce*. If you set bit 6 and the interface is the Active Controller, the interface remembers the response and asserts SRQ when control passes to another controller.

Since you can only control bit 6 of the serial poll response, only the bit corresponding to 64 in decimal of *hpib_rqst_srvce*'s response argument has affect. Thus:

```
hpib_rqst_srvce( eid, 64);
```

sets bit 6 of the interface's serial poll response and:

```
hpib_rqst_srvce( eid, 0);
```

clears it.

hpib_send_cmnd

The HP 27110A/B HP-IB and Series 550 Internal HP-IB interface cards send all the commands you specify with this routine, with odd parity. To do this, it overwrites the most significant bit of each command byte with a parity bit. This should not cause a problem since all HP-IB commands use only 7 bits, and the eighth is free for use as parity.

hpib_status_wait

The *hpib_status_wait* routine, when processing, holds off all other activity on that interface card. Other processes attempting to access the interface card will hang. It is strongly recommended that a non-zero timeout be in effect before calling *hpib_status_wait*.

hpib_wait_on_ppoll

The *hpib_wait_on_ppoll* routine, also, holds off all other activity on the interface card. Again, other processes attempting to access the interface card will hang and it is recommended that a non-zero timeout be in effect before calling *hpib_wait_on_ppoll*.

io_get_term_reason

Normally, this routine can indicate multiple reasons for a read termination by the values of the least significant three bits in its returned value:

Set Bit	Decimal	Meaning
(none)	0	Abnormal termination.
Bit 0	1	Number of bytes requested were read.
Bit 1	2	Specified termination character was detected.
Bit 2	4	Device-imposed termination condition was detected (e.g. EOI on HP-IB).

For example, if *io_get_term_reason* returns a 7 you know that the read terminated for three reasons: the byte count was reached, a **termination character** was seen, and a termination condition was detected.

The *io_get_term_reason* routine on the Series 500 has a limitation when a read is terminated for multiple reasons; it can only indicate one termination cause at a time. If a read terminates for multiple reasons, the value returned by *io_get_term_reason* is the value of the highest numbered reason. Thus, on the Series 500 the routine can only return a 0, 1, 2, or 4 (or a -1 if the routine itself fails). For instance, if a 4 is returned, you know that a device-imposed termination condition occurred, but you do not know if the byte count was reached or if a termination character was read as well.

On the Series 500, if you set a termination character for a GPIO interface that is using a 16-bit data path, only an 8-bit termination character is set (the least significant byte of the match value). During read operations, if the termination character is then seen as the lower byte in a data transfer, everything works correctly; both the upper and lower bytes of the transfer are received and the count of received bytes is incremented by two. However, if the termination character is seen as the upper byte of the transfer, both the upper and lower bytes are still read. The count of received bytes is only incremented by one though, indicating that the termination character was in the upper byte.

io_timeout_ctl

This routine allows you to set a time limit for I/O operations on an entity identifier associated with an interface file. The timeout value that you specify is a 32-bit long integer that indicates the length of the timeout in microseconds. However, the resolution of the effective timeout is system-dependent. On the Series 500 the timeout is rounded up to the nearest 10 millisecond boundary. For example, if you specify a timeout of 155000 microseconds (155 milliseconds), the effective timeout is rounded up to 160 milliseconds.

When an I/O operation is aborted due to a timeout *errno* is set to EIO. However, EIO is defined as **I/O error** and can be set by many other error conditions. On the Series 500, you can obtain more information by looking at the external HP-UX variable *errinfo*. When a timeout occurs, *errinfo* is set to the value 56.

io_speed_ctl

The Series 500 always provides DMA for the fastest possible I/O speeds. Therefore, *io_speed_ctl* has no affect on the Series 500.

io_width_ctl

Although this routine is designed to be used on any interface, the path width that you specify with it must be supported on the particular interface. On the Series 500, only the GPIO interface allows you to change data path widths and only two widths are currently supported: 8 bits and 16 bits. The routine returns an error if you access a GPIO interface with any width besides 8 or 16 bits or if you access any other interface with a width other than 8 bits.

Performance Tips

The performance of your I/O process on a Series 500 that uses DIL routines can be improved by following the basic guidelines listed below.

- Use buffers to hold data that you write to an interface. Transferring data that you have previously stored in a buffer is faster than if you specify the data string when you invoke the transfer. For example, the data transfer performed by the code:

```
int eid;          /*entity identifier descriptor*/
char buffer[12];  /*buffer to hold data*/

eid = io_open("/dev/raw_hpib", O_RDWR);
buffer = "data message"; /*store data in buffer*/
io_write( eid, buffer, 12); /*transfer data*/
```

is faster than the data transfer performed by the code:

```
int eid;          /*entity identifier descriptor*/
eid = io_open("/dev/raw_hpib", O_RDWR);
io_write( eid, "data message", 12); /*transfer data*/
```

- Make the number of bytes transferred divisible by the number of bytes per word that your system supports. Data transfers, both reading and writing, are faster if the number of bytes involved in the transfer falls on a word boundary. The Series 500 supports 4-byte words; therefore, the following code has an optimized performance because the byte counts are divisible by 4.

```
io_write( eid, buffer1, 12);
io_read( eid, buffer2, 40);
```

- If you are the super-user, you can use the *memlck* routine (see *HP-UX Reference: Section 2*) to lock your I/O process's address space into physical memory. Data transfer times are reduced because they are carried out directly from the user area and do not have to be first moved to the system area. However, you can not lock an arbitrarily large amount of space for your process since there is a point at which your system's performance will begin to degrade.
- For processes running with an effective user ID of super-user, it is possible to lock the process in memory with *plock(2)* (see *HP-UX Reference*). This lock is different than *memlck* (as mentioned above). *int plock(2)* informs the system that the process text, data, or both are not to be swapped out of memory. The following example illustrates the use of *plock*:

```
#include <sys/lock.h>
main()
{
    int plock();
    plock(PROCLOCK); /* lock text and data semnets into memory*/
    ...
    plock(UNLOCK); /* unlock my process*/
}
```

- Use auto-addressing for all read and write operations. (See *Chapter 3: Setting up Talkers and Listeners*.)

- Increasing the system priority of an I/O process can be accomplished by using *rtprio(2)*. *rtprio* requires the process to be running with an effective user *ID* of super-user. The real time priorities available with *rtprio* are non-degrading priorities. Caution must be observed when using real time priorities since one can increase their priority above system processes. This may cause undesirable behavior. For example, requesting a real time priority in the range of 0-63 places your process in a higher priority than the DIL interrupt handler system process. This means that interrupts could be lost if there is not sufficient CPU resource available. The following example places the calling process at the lowest (least important) real time priority:

```
#include <sys/rtprio.h>
main()
{
    int rtprio(), my_proc;

    my_proc = 0;          /* a zero process # tells rtprio to refer to the */
                        /* calling process. */
    rtprio(my_proc, 127); /* priority 127 = lowest real time priority*/
    ...
    ...
    ...
    rtprio(my_proc, RTPRIO_RT0FF); /* turn off real time priority*/
}
```


Character Codes

B

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

STD-LL-00182

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
\$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
-	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

Character Codes (cont.)

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

Index

a

Active Controller:	
Bus Management	7
Computer Role	36
Determining	40
Functions of	40
Passing Control	55,56
Addressed Commands	31,32
Addresses:	
Bus	13,14
Listen	31,32
Talk	31,32
Addressing:	
Auto-addressing	41
Listeners and Talkers	41
Manual	41
AND-ing	51
Asserting Lines	6
ATN (Attention)	7,31,57,69
Auto-addressing Bus	41

b

buf	68,70
Buffering I/O Operations	68-76
Bus Address	13,14,82

c

C Routines:	
close	3
open	2,3
read	11,17
write	11,17
C:	
Linking DIL	2
Onionskin	3
Call by Reference	3
Call by Value	3
Character Special File	13
Clear	47
close	3,16
Commands:	
Addressed	31,32

Secondary	31,32
Sending HP-IB Commands	38-39
Talk and Listen Addresses	31,32
Universal	31,32
Communication Using Special Files	12-14
Compatibility:	
Data	5
Electrical and Mechanical	5,6
Timing	5,6
Compiling	2
Computer's Role on HP-IB	36
Controller:	
Active	36
Description (HP-IB Device)	6
Non-Active	36
System	36
Controlling:	
GPIO Interface	77-82
HP-IB Interface	31-74
I/O Parameters	20-25
Interrupts	27
Path Width	79
Transfer Speed	79
count	68,70
Creating:	
GPIO Interface File	14,75
HP-IB Interface File	13
Interface Files	12,81
CTL0	76-79
CTL1	76-79

d

Data-In	9
Data:	
Compatibility	5
Controlling Path Width	79
Controlling Transfer Speed	79
End of Data Transfer	23-24
Handshaking Methods	9
Holding Valid	6,9
Performing Transfers	77
Reading	9
Setting Path Width	21-22
Setting Transfer Speed	22
Timeout on Transfer	20-21
Transferring by Active Controller	44
Writing	9
DAV (Data Valid)	6

DCL	27
Designing Error Checking Routines	18
Determining:	
Active Controller	41
Bus Address of the Interface Card	82
Controller's Status	60
System Controller	57
The Driver	81
The Select Code	81
When You are Addressed	66
Why a Read Terminated	25-26,80
DEVICE CLEAR	32,33,45
Devices:	
Clearing HP-IB	45
Remote Control of	43
Triggering	44
DIL (Device I/O Library):	
Introduction to	1,2
DIL Routines:	
Calling from C	2
Calling from FORTRAN	3
Calling from Pascal	2
General Purpose	11
GPIO Specific	76
Introduction to	1-3
Library Containing	2
Linking from C, FORTRAN, Pascal	2
Location of	2
Restrictions	82
Role Designations	36
Used to Control I/O Parameters	20-24
DMA (Direct Memory Access)	22,79,85
DOUT CLEAR (Data Out Lines Clear)	77
Driver	12-14,81
dvio.h	68

e

EBADF	39,51,54,55,58,60,62,65
eid:	
Description of	15
Using	15,19
EINVAL	62,65,79
EIO	2,39,51,54,55,58,64,85
EIR (External Interrupt Request Line)	8,76
Electric Logic Levels	5
Electrical Compatibility	5
Enabling Local Control	44
ENOENT	19

ENOTTY	39,51,54,55,58,60,62,65
Entity Identifier	15,82
EOI (End or Identify)	7,21,29,69
errinfo	85
errno.h	18
errno:	
Description of	18
Using	18
Values of	39,51,54,55,58,60,62,65
Errors:	
During hpib_abort and hpib_ren_ctl	58
During Parallel Polling	50-51
I/O	85
Locating In Buffered I/O Operations	74
While Configuring Response	62
While Passing Control	56
While Requesting Service	61
While Sending Commands	39
While Waiting on Status	65

f

fcntl.h	15,16
Files:	
Closing	3,15
Creating Interface Files	12-14
dvio.h	68
errno.h	18
fcntl.h	15,16
Non-Raw HP-IB Device File	41
Opening	2,15
Raw HP-IB Device File	12-14
Reading and Writing	17,18
Special	12-14
FLAG Switches	9
Floating Lines	6
Fork	21,24,25
FORTTRAN:	
\$Alias Directive	3
Call by Value	3
Calling DIL Routines	3
Linking DIL	2
Full-Mode Handshake	9

g

GET	27
GO TO LOCAL	32,34,43
GPIO (General Purpose Input/Output):	
Configuring the Interface	75
Controlling Data Path Width	79
Controlling the Interface	76
Creating the Interface	75
Data Lines	8
Handshake Lines	8
HP 27110A Interface	81-83
Interface Description	8
Introduction to	1,8
Performing Data Transfers	77
Resetting the Interface	77
Special Purpose Lines	8,78
Specific DIL Routines	76
gpio_get_status	76,78
gpio_set_ctl	76,78

h

Handshaking:	
Description of	5,9
GPIO Lines	8
HP-IB Lines	6
Types of	9
Hanging a Program	20,21
HP 27110A GPIO Interface	81-83
HP 27110A/B HP-IB Interface	81-83
HP-IB (Hewlett-Packard Interface Bus):	
Bus Commands	31-34
Bus Management Lines	7
Computer's Role	36
Controlling	31-75
Devices	6
DIL Routines	35,36
General Structure	6
Handshake Lines	6
HP 27110A/B Interface	81-83
Internal 550 HP-IB Interface	81-83
Introduction to	1,6-7
Non-Raw Device Files	41
Raw Device Files	12-14
HPIBATN	69-73
HPIBCHAR	69-73
HPIBEOI	69-73
HPIBREAD	69-73

HPIBWRITE	69-73
hpib_abort	35,36,55,57
hpib_bus_status	35,36,40,42,46,55,58,59,63-66,82
hpib_card_ppoll_resp	35,36,61,83
hpib_eoi_ctl	35,36
hpib_io	35,36,68,71-73
hpib_pass_ctl	35,36,55
hpib_ppoll	35,36,47,49,50
hpib_ppoll_resp_ctl	35,36,63
hpib_ren_ctl	35,36,57,58
hpib_rqst_srvce	35,36,60,83
hpib_send_cmnd	31,32,35,36,38-42,44,83
hpib_spoll	35,36,47,54
hpib_status_wait	35,36,46,51,63,65,84
hpib_wait_on_ppoll	35,36,51-53,84

i

I/O (Input/Output Line)	8,76,77,80
IFC (Interface Clear Line)	7,19,27,57
Input/Output:	
Buffering Operations	68-76
Controlling Parameters	20-25
Errors	55
Operation Template	68-70
Setting Timeout	20-21
Interface:	
Additional Functions	5
Background Information on	4-5
Bus Address	82
Closing Files	16
Compatibilities	5
Creating Interface Files	12-14
Functional Diagram of	4
HP 27110A(GPIO)	81-83
HP 27110A/B(HP-IB)	81-83
Internal 550 HP-IB	81-83
Opening Files	15
Opening HP-IB Interface File	37
Primary Function of	4-5
Resetting	19,77
Setting Switches	75
Why Its Needed	4-5
Internal 550 HP-IB Interface	81-83
Interrupts:	
Conditions	27
Controlling Routines	27,80
Description of	27-29
Handlers	27,80

iodetail:

Allocating Space for Structure	70
Buf	68,70
Count	68,70
Mode	68-69
Template	68-70
Terminator	68,70
io_alloc	70
io_eol_ctl	11,20,23,24,80
io_get_term_reason	11,20,25,84
io_interrupt_ctl	11,28,29,80
io_on_interrupt	11,27,28,63,65,67,80
io_reset	11,19,55,77,79
io_speed_ctl	11,20,22,79,85
io_timeout_ctl	11,20,21,46,50,54,64,85
io_width_ctl	11,20-22,79,85

k

k-bytes	22
---------------	----

l

Libraries:

/lib/libXXX.a	2
/usr/lib/libdvio.a	2
/usr/lib/libXXX.a	2
Linking	2
Location of	2

Lines:

Asserting	6
Floating	6
GPIO Data	8
GPIO Handshaking	8
GPIO Special Purpose	8,78
HP-IB Bus Management Contol	7
HP-IB Handshaking	6
Sensing	6

Linking DIL Routines:

To C Program	2
To FORTRAN Program	2
To Pascal Program	2

Listeners:

Description (HP-IB Device)	6
Setting up	41

Local Control:

Enabling	43
Locking Out	43

LOCAL LOCKOUT	32,33,43
Local:	
State	43
Switch	43
Locking Out Local Control	44
Logic Levels	5
LTN	27

m

Major Number:	
Determining	12,81
Using	12-14
Manually Addressing Bus	41
mask:	
Calculating	51-53
Parameter Description	27,28
Match	23
Matching Levels	5
Mechanical Compatibility	5
Minor Number:	
Determining	12,81
Using	12-14
mknod	12-14,75,81
MLA (My Listen Address)	42
mode	68,69
MTA (My Talk Address)	42,44

n

NDAC (Not Data Accepted)	6,59
Non-Active Controller:	
Accepting Active Control	64,65
Computer Role	36
Description of	55
Determining When Addressed	66
Disabling Parrallel Poll Response	64
Errors While Requesting Service	62
Receiving Control	55
Requesting Service	61,62
Responding to Parallel Polls	62,63
Non-Raw HP-IB Device File	41
NO_FILE	16
NRFD (Not Ready For Data)	6

O

Onionskin	3
open	2,15
Opening:	
HP-IB Interface Files	37
Interface Files	15
OR-ing	51,67,68,82
O_RDWR:	
Description of	15
Using	15

P

PARALLEL POLL CONFIGURE	32,34,47,49
PARALLEL POLL DISABLE	32,34,49
PARALLEL POLL ENABLE	32,34,47,48
PARALLEL POLL UNCONFIGURE	32
Parallel Polling:	
Conducting	49,50
Configuring Responses	47-49
Disabling Responses	49,63
Errors During	50,51
Responding to	61
Waiting for Response	51-53
Pascal:	
Call by reference	3
Calling DIL Routines	2
Linking DIL	2
Passing Parameters:	
By Reference	3
By Value	3
Pathname	12,14
PCTL (Peripheral Control Line)	8,76,77
Performance Tips Series 500	86
PFLG (Peripheral Flag Line)	8,76
PPOLL	27
PRESET (Peripheral Reset Line)	19,77
PSTS (Peripheral Status Line)	75,76,80
Pulse-Mode Handshake	9

R

Raw:	
Definition	13
HP-IB Device File	13
Mode	12,13
Read Termination Character:	
Description of	23
Removing	24
Setting	23,24,80
read:	
System Routine	11,17,77
Why Terminated	25,80
Recognizing Devices	33
Remote State	43
Removing Read Termination Characters	24,25
REN (Remote Enable)	7,27,31,43,56,57
Requesting Service	60,61
Resetting Interfaces	19,77
Routines:	
C	2,3,11,17
DIL General Purpose	11
Error Checking	18
For Calling I/O Parameters	20
GPIO Specific	76
Service	47
System	11

S

Secondary Commands	31,32
Select Code	13,14,81
SELECTED DEVICE CLEAR	32,33,45
Sending HP-IB Commands	38,39
sense Value	51-53
sense,Calculating	52
Sensing Lines	6
SERIAL POLL DISABLE	32,33
SERIAL POLL ENABLE	32,33
Serial Polling:	
Conducting	54
Description of	53
Errors During	54
Service Request:	
Description of	46
SRQ Line	46
Using Service Routine	46,47
Service Routine	53
Servicing Requests	46,47

Setting:

Data Path Width	21,22
GPIO Interface	75
I/O Timeout	20
Interface Switches (GPIO Interface)	77
Read Termination Character	23,24
Transfer Speed	22,80
Up Talkers and Listeners	42
SIE0	27
SIE1	27
Special Files:	
/dev/hpib	37
/dev/raw_gpio	14
/dev/raw_hpib	13,15,17,37
/dev/raw_hpib1	14
/dev/raw_hpib2	14
Character	13
Description	12
Directory Containing	12
Non-Raw HP-IB Device File	41
Raw HP-IB Device File	13
Special Purpose Lines	8,78
SRQ (Service Request)	7,27,46,47,51,54,57,59,60,64,82,83
Status Byte Message	53
STI0	76,78
STI1	76,78
System Controller:	
Bus Management	7
Computer Role	36
Description of	56-58
Determining	56
Duties	56
System Routines	11

t

TAKE CONTROL	32
Talk Address	31,32
Talk and Listen Addresses	31,32
Talkers:	
Description (HP-IB Device)	6
Setting up	41
TCT	27
Terminated Read	25,80
Termination Character:	
Description of	23
Removing	24
Setting	23,24,80
terminator	68,70

Timeouts	20,21,79
Timing Compatibility	5
TLK	27
TRIGGER	32,33,44

u

Unaddressing	33
Universal Commands	31,32
UNLISTEN	32,33,38,39,41,44
UNTALK	32,33,38,39

v

Valid Data	6,9
------------------	-----

w

write	11,17,77
-------------	----------

x

XOR-ed	5
--------------	---

Table of Contents

Lex: A Lexical Analyzer Generator

Introduction	1
Lex Source	3
Lex Regular Expressions	4
Operators	4
Character classes	5
Arbitrary character	5
Optional expressions	6
Repeated expressions	6
Alternation and Grouping	6
Context sensitivity	7
Repetitions and Definitions	7
Lex Actions	8
Example	9
Ambiguous Source Rules	11
Lex Source Definitions	13
Usage	14
HP-UX	14
Lex and Yacc	15
Examples	15
Left Context Sensitivity	18
Character Set	20
Summary of Source Format	21
Caveats and Bugs	22



Lex

A Lexical Analyzer Generator

Introduction

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C. Lex itself exists on HP-UX, but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source*) into the host general-purpose language. The generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input*) and perform the specified actions for each expression as it is detected. See Figure 1.

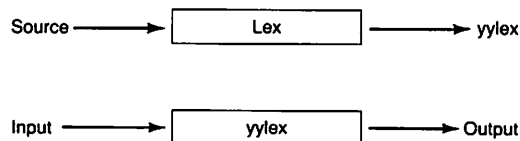


Figure 1: An overview of Lex

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%  
[\\t]+$
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \\t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the \$ indicates "end of line," similar to ED. No action is specified, so the program generated by Lex (*yylex*) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%  
[\\t]+$ ;  
[\\t]+ printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

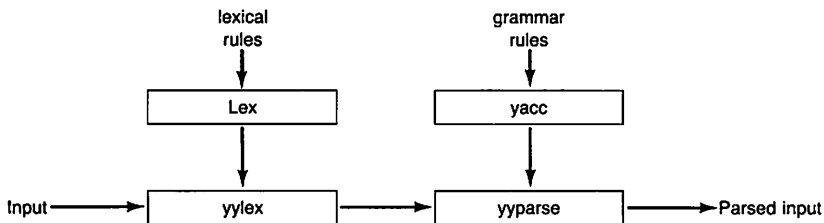


Figure 2: Lex with Yacc

Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered, as cases of a switch statement in C. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before "*cd...*" Such backup is more costly than the processing of simpler languages.

Lex Source

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function **printf** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*. A way of dealing with this will be described later.

Lex Regular Expressions

The definitions of regular expressions are similar to those in ED. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + ! ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\ . Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes

Classes of characters can be specified using the operator pair []. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are `\`, `-` and `^`. The `-` character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using `-` between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., `[0-z]` in ASCII is many more characters than it is in EBCDIC.) If it is desired to include the character `-` in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the `^` operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except *a*, *b*, or *c*, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The `\` character provides the usual escapes within character class brackets.

Arbitrary character

To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions

The operator `?` indicates an optional element of an expression. Thus

`ab?c`

matches either `ac` or `abc`.

Repeated expressions

Repetitions of classes are indicated by the operators `*` and `+`.

`a*`

is any number of consecutive `a` characters, including zero; while

`a+`

is one or more instances of `a`. For example,

`[a-z]+`

is all strings of lower case letters. And

`[A-Za-z][A-Za-z0-9]*`

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator `|` indicates alternation:

`(ab|cd)`

matches either `ab` or `cd`. Note that parentheses are used for grouping, although they are not necessary on the outside level;

`ab|cd`

would have sufficed. Parentheses can be used for more complex expressions:

`(ab|cd+)?(ef)*`

matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.

Context Sensitivity

Lex will recognize a small amount of surrounding context. The two simplest operators for this are `^` and `$`. If the first character of an expression is `^`, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of `^`, complementation of character classes, since that only applies within the `[]` operators. If the very last character is `$`, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the `/` operator character, which indicates trailing context. The expression

```
ab / cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab $
```

is the same as

```
ab / \n
```

Left context is handled in Lex by *start conditions* as explained in the section on left context sensitivity. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered “being at the beginning of a line” to be start condition *ONE*, then the `^` operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions

The operators `{ }` specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

Finally, initial `%` is special, being the separator for Lex source segments.

Lex Actions

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, “;” as an action causes this result. A frequent rule is

```
[\t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "      ;
"\t"    ;
"\n"    ;
```

with the same result, although in different style. The quotes around `\n` and `\t` are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like `[a-z]+`. Lex leaves this text in an external character array named `yytext`. Thus, to print the name found, a rule like

```
[a-z]+ printf("%s", yytext);
```

will print the string in `yytext`. The C function `printf` accepts a format argument and data to be printed; in this case, the format is “print string” (% indicating data conversion, and `s` indicating string type), and the data are the characters in `yytext`. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+ ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches `read` it will normally match the instances of `read` contained in `bread` or `readjust`, to avoid this, a rule of the form `[a-z]+` is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count `yylen` of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+ {words++; chars += yylen;};
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yylen-1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look-ahead offered by the / operator, but in a different form.

Example

Consider a language which defines a string as a set of characters between quotation marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\["^"]*      {
    if (yytext[yylen-1] == '\\')
        yymore();
    else
        ... normal user processing
}
```

which will, when faced with a string such as "abc\ "def" first match the five characters "abc\ "; then the call to *yymore()* will cause the next part of the string, "def", to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "= -a". Suppose it is desired to treat this as "= - a" but print a message. A rule might be

```
=[a-zA-Z]    {
    printf("Operator (=) ambiguous\n");
    yyless(yylen-1);
    ... action for =- ...
}
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "= -". Alternatively it might be desired to treat this as "= - a". To do this, just return the minus sign as well as the letter to the input:

```
=[a-zA-Z]    {
    printf("Operator (=) ambiguous\n");
    yyless(yylen-2);
    ... action for =- ...
}
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-[A-Za-z]
```

in the first case and

```
=/[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of “= - 3”, however, makes

```
=-/[^\t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1. **input()** which returns the next input character;
2. **output(c)** which writes the character *c* on the output; and
3. **unput(c)** pushes the character *c* back onto the input stream to be read later by **input()**.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex look-ahead will not work.

Lex does not look ahead at all if it does not have to, but every rule ending in +, \, *, ?, or \$ or containing / implies look-ahead. Look-ahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

Ambiguous Source Rules

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+ identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because $[a-z]^+$ matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the `.` operator will not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like $[\backslash n]^+$ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some Lex rules to do this might be

```
she      s++;
he       h++;
\n       ;
.        ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that `.` does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means “go do the next alternative.” It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```

she      {s++; REJECT;}
he       {h++; REJECT;}
\n       |
.        |

```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible *a priori* to tell which input characters were in both classes.

Consider the two rules

```

a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}

```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```

%%
[a-z][a-z]      {digram[yytext[0]][yytext[1]]++; REJECT;}
.               |
\n              |

```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Lex Source Definitions

Remember the format of the Lex source:

```
{definitions}  
%%  
{rules}  
%%  
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.
3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D           [0-9]  
E           [DEde][-+]?{D}+  
%%  
{D}+       printf("integer");  
{D}+,"{D}*({E})?  
           |  
{D}*","{D}+({E})?  
           |  
{D}+{E}   printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/","EQ      printf("integer");
```

could be used in addition to the normal rule for integers. The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

Usage

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

HP-UX

The library is accessed by the loader flag *-ll* for C, so an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input*, *output* and *unput* are given, the library can be avoided.

Lex and Yacc

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the HP-UX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

Examples

As a simple example, consider copying an input file while adding 3 to every positive number which is divisible by 7. Here is a suitable Lex source program

```
%Z
int k;
[0-9]+ {
    k = atoi(yytext);
    if (k%7 == 0)
        printf("%d", k+3);
    else
        printf("%d", k);
}
```

to do just that. The rule `[0-9]+` recognizes strings of digits; `atoi` converts the digits to binary and stores the result in `k`. The operator `%` (remainder) is used to check whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%Z
int k;
-?[0-9]+ {
    k = atoi(yytext);
    printf("%d", k%7 == 0 ? k+3 : k);
}
-?[0-9.]+ ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```


Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*."

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```

        int lensg[100];
ZZ
[a-z]+ lensg[yyleng]++;
.      |
\n     |
ZZ
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
    if (lensg[i] > 0)
        printf("%5d%10d\n",i,lensg[i]);
return(1);
}

```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1)*; indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some program fragments which converts double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```

a      [aA]
b      [bB]
c      [cC]
...
z      [zZ]

```

An additional class recognizes white space:

```

W      [\t]*

```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```

{d}{o}{u}{b}{l}{e}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]='d'? "real" : "REAL");
}

```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```

^"      "[^ 0] ECHO;

```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of `^`. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      |
[0-9]+{W}"{W}{d}{W}[+-]?{W}[0-9]+  |
","{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ {
/* convert constants */
for(p=yytext; *p != 0; p++)
{
    if (*p == 'd' | *p == 'D')
        *p+= 'e' - 'd';
    ECHO;
}
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D`. The program then adds `'e' - 'd'` which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial `d`. By using the array `yytext` the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}      |
{d}{c}{o}{s}      |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial `d` changed to initial `a`:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
    yytext[0] += 'a' - 'd';
    ECHO;
}
```

And one routine must have initial `d` changed to initial `r`:

```
{d}{i}{m}{a}{c}{h}      {yytext[0] += 'r' - 'd';
                          ECHO;
                          }
```

To avoid such names as `dsinx` being detected as instances of `dsin`, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.                      ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The `^` operator, for example, is a prior context operator, recognizing immediately preceding left context just as `$` recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
int flag;

%%
^a    {flag = 'a'; ECHO;}
^b    {flag = 'b'; ECHO;}
^c    {flag = 'c'; ECHO;}
\n    {flag = 0; ECHO;}
magic {
    switch (flag)
    {
    case 'a': printf("first"); break;
    case 'b': printf("second"); break;
    case 'c': printf("third"); break;
    default: ECHO; break;
    }
}
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the `<>` brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the `<>` prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic    printf("first");
<BB>magic    printf("second");
<CC>magic    printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

Character Set

The programs generated by Lex handle character I/O only through the routines *input*, *output*, and *unput*. Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant 'a'. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

```
%T
 1      Aa
 2      Bb
...
26     Zz
27     \n
28     +
29     -
30     0
31     1
...
39     9
%T
```

Sample character table.

Summary of Source Format

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form “name space translation”.
2. Included code, in the form “space code”.
3. Included code, in the form

```
%{
code
}%
```

4. Start conditions, given in the form

```
%S name1 name2 ...
```

5. Character set tables, in the form

```
%T
number space character-string
...
%T
```

6. Changes to internal array sizes, in the form

```
%x nnn
```

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

<i>Letter</i>	<i>Parameter</i>
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form “expression action” where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

<code>x</code>	the character "x."
<code>"x"</code>	an "x", even if x is an operator.
<code>\x</code>	an "x", even if x is an operator.
<code>[xy]</code>	the character x or y.
<code>[x-z]</code>	the characters x, y or z.
<code>[^x]</code>	any character but x.
<code>^x</code>	an x at the beginning of a line.
<code><y>x</code>	an x when Lex is in start condition y.
<code>x\$</code>	an x at the end of a line.
<code>x?</code>	an optional x.
<code>x*</code>	0,1,2, ... instances of x.
<code>x+</code>	1,2,3, ... instances of x.
<code>x y</code>	an x or a y.
<code>(x)</code>	an x.
<code>x/y</code>	an x but only if followed by y.
<code>{xx}</code>	the translation of xx from the definitions section.
<code>x{m;n}</code>	m through n occurrences of x.

Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

Table of Contents

Yacc: Yet Another Compiler-Compiler

Introduction	2
1: Basic Specifications	4
2: Actions	6
3: Lexical Analysis	8
4: How the Parser Works	9
5: Ambiguity and Conflicts	13
6: Precedence	17
7: Error Handling	19
8: The Yacc Environment	21
9: Hints for Preparing Specifications	22
Input Style	22
Left Recursion	22
Lexical Tie-ins	23
Reserved Words	24
10: Advanced Topics	24
Simulating Error and Accept in Actions	24
Accessing Values in Enclosing Rules	24
Support for Arbitrary Value Types	25
References	26
Appendix A: A Simple Example	27
Appendix B: Yacc Input Syntax	29
Appendix C: An Advanced Example	31
Appendix D: Old Features Supported but Not Encouraged	36



Yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the “*lexical analyzer*”) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called “*grammar rules*”; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C¹ and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma “,” is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a “*terminal symbol*”, while the structure recognized by the parser is called a “*nonterminal symbol*”. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
. . .
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc’s ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as “,” must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be “slipped in” to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere^{2 3 4}. Yacc has been extensively used in numerous practical applications, including *lint*⁵, the Portable C Compiler⁶, and a system for typesetting mathematics⁷.

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, “(grammar) rules”, and *programs*. The sections are separated by double percent “%%” marks. (The percent “%” is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot “.”, underscore “_”, and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes. As in C, the backslash “\” is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'  newline
'\r'  return
'\'   single quote “'”
'\'   backslash “\”
'\t'  tab
'\b'  backspace
'\f'  form feed
'\xxx' “xxx” in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar “|” can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A      :      B C D ;
A      :      E F ;
A      :      G ;
```

can be given to Yacc as

```
A      :      B C D
      |      E F
      |      G
;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 . . .
```

in the declarations section. (See Sections 3, 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as “end-of-file” or “end-of-record”.

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A      :      '( B )'
          {      hello( 1, "abc" ); }
```

and

```
XXX    :      YYY ZZZ
          {      printf("a message\n");
                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "\$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "\$\$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2, . . . , which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A      :      B C D ;
```

for example, then \$2 has the value returned by C, and \$3 the value returned by D.

As a more concrete example, consider the rule

```
EXPR  :      '( EXPR )' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
EXPR  :      '( EXPR )'      { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it (\$1). Thus, grammar rules of the form

```
A      :      B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```

A      :      B
          { $$ = 1; }
        C
          { x = $2; y = $3; }
;

```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```

$ACT   :      /* empty */
          { $$ = 1; }
;

A      :      B $ACT C
          { x = $2; y = $3; }
;

```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label *L*, and descendants *n1* and *n2*, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```

EXPR   :      EXPR '+' EXPR
          { $$ = node('+', $1, $3); }

```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks “%{” and “%}”. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in “*yy*”; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the “*token number*”, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the “# define” mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
    extern int yyval;
    int c;
    . . .
    c = getchar();
    . . .
    switch( c ) {
        . . .
    case '0':
    case '1':
        . . .
    case '9':
        yyval = c-'0';
        return( DIGIT );
        . . .
    }
    . . .
}
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk⁸. These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. *Lex* can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The “*current state*” is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yyllex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a “.”) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
. reduce 18
```

refers to *grammar rule 18*, while the action

```
IF shift 34
```

refers to *state 34*.

Suppose the rule being reduced is

```
A : x y z ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a *goto*. In any case, the uncovered state contains an entry such as:

```
A goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action “turns back the clock” in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yyval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables \$1, \$2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme :      sound place
      ;
sound  :      DING DONG
      ;
place  :      DELL
      ;
```

When Yacc is invoked with the `-v` option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
  $accept : _rhyme $end

  DING shift 3
  , error

  rhyme goto 1
  sound goto 2

state 1
  $accept : rhyme_$end

  $end accept
  , error

state 2
  rhyme : sound_place

  DELL shift 5
  , error

  place goto 4

state 3
  sound : DING_DONG

  DONG shift 6
  , error

state 4
  rhyme : sound place_ (1)

  , reduce 1

state 5
  place : DELL_ (3)

  , reduce 3

state 6
  sound : DING DONG_ (2)

  , reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The `_` character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is “shift 3”, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is “shift 6”, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is “shift 5”, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by “\$end” in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as “*DING DONG DONG*”, “*DING DONG*”, “*DING DONG DELL DELL*”, etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

`EXPR : EXPR '-' EXPR`

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

`EXPR - EXPR - EXPR`

the rule allows this input to be structured as either

`(EXPR - EXPR) - EXPR`

or as

`EXPR - (EXPR - EXPR)`

(The first is called “*left association*”, the second “*right association*”.)

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

`EXPR - EXPR - EXPR`

When the parser has read the second `expr`, the input that it has seen:

`EXPR - EXPR`

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to `expr` (the left side of the rule). The parser would then read the final part of the input:

`- EXPR`

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

`EXPR - EXPR`

it could defer the immediate application of the rule, and continue reading the input until it had seen

`EXPR - EXPR - EXPR`

It could then apply the rule to the rightmost three symbols, reducing them to `expr` and leaving

`EXPR - EXPR`

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
EXPR - EXPR
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a “*shift / reduce conflict*”. It may also happen that the parser has a choice of two legal reductions; this is called a “*reduce / reduce conflict*”. Note that there are never any “Shift/shift” conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a “*disambiguating rule*”.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an “if-then-else” construction:

```
stat      :      IF '(' cond ')' stat_  
          ;      IF '(' cond ')' stat_ELSE stat  
          ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {  
    IF ( C2 ) S1  
}  
ELSE S2
```

or

```
IF ( C1 ) {  
    IF ( C2 ) S1  
    ELSE S2  
}
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding “un-*ELSE*’d” *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (*-v*) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

stat : IF ( cond ) stat_      (18)
stat : IF ( cond ) stat_ELSE stat

ELSE   shift 45
      ,   reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by “.”, is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF ((' cond ') stat
```

Once again, notice that the numbers following “shift” commands refer to other states, while the numbers following “reduce” commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references²³⁴ might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
EXPR : EXPR OP EXPR
```

and

```
EXPR : UNARY EXPR
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'
```

```
%%
```

```
EXPR :      EXPR '=' EXPR
      :      EXPR '+' EXPR
      :      EXPR '-' EXPR
      :      EXPR '*' EXPR
      :      EXPR '/' EXPR
      :      NAME
      ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( (c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

EXPR :      EXPR '+'  EXPR
      |      EXPR '-'  EXPR
      |      EXPR '*'  EXPR
      |      EXPR '/'  EXPR
      |      '-' EXPR  %prec '*'
      |      NAME
      ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially “cookbook” fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser “restarted” after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name “error” is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token “error” is legal. It then behaves as if the token “error” were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat      :      error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat      :      error 'i'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error '\n' { printf( "Reenter last line: " ); } input
      { $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrork ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input : error '\n'
      { yyerrork;
        printf( "Reenter last line: " ); }
      input
      { $$ = $4; }
      ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error
     { resynch();
       yyerrork ;
       yyclearin ; }
     ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a `-ly` argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
    return( yyparse() );
}
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
    fprintf( stderr, "%s\n", s );
}
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name      :      name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list      :      item  
          |      list ',' item  
          ;
```

and

```
seq       :      item  
          |      seq item  
          ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq       :      item  
          |      item seq  
          ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable. It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq      :      /* empty */ ! seq item
          ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
    int dflag;
%}
... other declarations ...

%%

prog  :      decls stats
      ;

decls :      /* empty */
          {      dflag = 1; }
      ;      decls declaration
      ;

stats :      /* empty */
          {      dflag = 0; }
      ;      stats statement
      ;

... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of “backdoor” approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyperror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent      :      adj noun verb adj noun
           { look at the sentence . . . }
           ;

adj       :      THE      { $$ = THE; }
           |      YOUNG   { $$ = YOUNG; }
           . . .
           ;

noun      :      DOG      { $$ = DOG; }
           |      CRONE   { if( $0 == YOUNG ){
                           printf( "what?\n" );
                           }
                           $$ = CRONE; }
           ;
           . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a \$\$ or \$n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*⁵ will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
    body of union ...
}
```

This declares the Yacc value stack, and the external variables *lval* and *yyval*, to have type equal to this union. If Yacc was invoked with the *-d* option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a *typedef* used to define the variable *YYSTYPE* to represent this union. Thus, the header file might also have said:

```
typedef union {
    body of union ...
} YYSTYPE;
```

The header file must be included in the declarations section, by use of *%{* and *%}*.

Once *YYSTYPE* is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords *%token*, *%left*, *%right*, and *%nonassoc*, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, *%type*, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no “*a priori*” type. Similarly, reference to left context values (such as *\$0* – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between *<* and *>*, immediately after the first *\$*. An example of this usage is

```
rule      :      aaa { $<intval>$ = 3; } bbb
           ;
           {      fun( $<intval>2, $<other>0 ); }
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of *%type* will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of *\$n* or *\$\$* to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, “LR Parsing,” *Comp. Surveys* 6(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, “Deterministic Parsing of Ambiguous Grammars,” *Comm. Assoc. Comp. Mach.* 18(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, “Lint, a C Program Checker,” *Comp. Sci. Tech. Rep. No. 65* (December 1977).
6. S. C. Johnson, “A Portable Compiler: Theory and Practice,” *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Charry, “A System for Typesetting Mathematics,” *Comm. Assoc. Comp. Mach.* 18 pp. 151-157 (March 1975).
8. M. E. Lesk, “Lex – A Lexical Analyzer Generator,” *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, Murray Hill, New Jersey (October 1975). (See *HP-UX Concepts and Tutorials*, Vol. 1.)

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */

%% /* beginning of rules section */

list : /* empty */
      | list stat '\n'
      | list error '\n'
      { yyerror; }
      ;

stat : expr
      | LETTER '=' expr
      { regs[$1] = $3; }
      ;
```

```

expr : '(' expr ')'
      {
        $$ = $2; }
| expr '+' expr
      {
        $$ = $1 + $3; }
| expr '-' expr
      {
        $$ = $1 - $3; }
| expr '*' expr
      {
        $$ = $1 * $3; }
| expr '/' expr
      {
        $$ = $1 / $3; }
| expr '%' expr
      {
        $$ = $1 % $3; }
| expr '&' expr
      {
        $$ = $1 & $3; }
| expr '|' expr
      {
        $$ = $1 | $3; }
| '-' expr
      {
        %prec UMINUS
        $$ = - $2; }
| LETTER
      {
        $$ = regs[$1]; }
| number
;

number : DIGIT
        {
          $$ = $1;   base = ($1==0) ? 8 : 10; }
| number DIGIT
        {
          $$ = base * $1 + $2; }
;

%% /* start of Programs */

yylex() { /* lexical analysis routine */
through 25 /* returns LETTER for a lower case letter, yylval = 0 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

  int c;

  while( (c=getchar()) == ' ' ) { /* skip blanks */ }

  /* c is now nonblank */

  if( islower( c ) ) {
    yylval = c - 'a';
    return ( LETTER );
  }
  if( isdigit( c ) ) {
    yylval = c - '0';
    return( DIGIT );
  }

  return( c );
}

```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
/* grammar for the input to Yacc */

/* basic entities */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by
colon */
%token NUMBER /* [0-9]+ */

/* reserved words: %type => TYPE, %left => LEFT, etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the %{ mark */
%token RCURL /* the %} mark */

/* ascii character literals stand for themselves */

%start spec

%%

spec : defs MARK rules tail
    ;

tail : MARK { In this action, eat up the rest of the file }
    ; /* empty: the second MARK is optional */

defs : /* empty */
    ; defs def
    ;

def : START IDENTIFIER
    ; UNION { Copy union definition to output }
    ; LCURL { Copy C code to output file } RCURL
    ; ndefs rword tag nlist
    ;

rword : TOKEN
    ; LEFT
    ; RIGHT
    ; NONASSOC
    ; TYPE
    ;
```

```

tag      :      /* empty: union tag is optional */
          |      '<' IDENTIFIER '>'
          ;

nlist    :      nmno
          |      nlist nmno
          |      nlist ',' nmno
          ;

nmno     :      IDENTIFIER          /* NOTE: literal illegal with %type */
          |      IDENTIFIER NUMBER /* NOTE: illegal with %type */
          ;

/* rules section */

rules    :      C_IDENTIFIER rbody prec
          |      rules rule
          ;

rule     :      C_IDENTIFIER rbody prec
          |      '|' rbody prec
          ;

rbody    :      /* empty */
          |      rbody IDENTIFIER
          |      rbody act
          ;

act      :      '{ { Copy action, translate $$, etc. } }'
          ;

prec     :      /* empty */
          |      PREC IDENTIFIER
          |      PREC IDENTIFIER act
          |      prec '|'
          ;

```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations $+$, $-$, $*$, $/$, unary $-$, and $=$ (assignment), and has 26 floating point variables, “a” through “z”. Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y . There are 26 interval valued variables “A” through “Z” that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 - 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the “,” is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{
#include <stdio.h>
#include <ctype.h>

typedef struct interval {
    double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];
%}

%start    lines

%union {
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG      /* indices into dreg, vreg arrays */
%token <dval> CONST         /* floating point constant */
%type <dval> dexp           /* expression */
%type <vval> vexp           /* interval expression */

/* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS               /* precedence for unary minus */

%%

lines : /* empty */
      ;
      lines line
      ;
```

```

line : dexp '\n'
      { printf( "%15.8f\n", $1 ); }
| vexp '\n'
      { printf( "(%15.8f, %15.8f)\n", $1.lo, $1.hi ); }
| DREG '=' dexp '\n'
      { dreg[$1] = $3; }
| VREG '=' vexp '\n'
      { vreg[$1] = $3; }
| error '\n'
      { yyerrok; }
;

dexp : CONST
| DREG
      { $$ = dreg[$1]; }
| dexp '+' dexp
      { $$ = $1 + $3; }
| dexp '-' dexp
      { $$ = $1 - $3; }
| dexp '*' dexp
      { $$ = $1 * $3; }
| dexp '/' dexp
      { $$ = $1 / $3; }
| '-' dexp
      { Zprec UMINUS
        { $$ = - $2; }
      }
| '(' dexp ')'
      { $$ = $2; }
;

vexp : dexp
      { $$,hi = $$,lo = $1; }
| '(' dexp ',' dexp ')'
      {
        $$,lo = $2;
        $$,hi = $4;
        if( $$,lo > $$,hi ){
          printf("interval out of order\n");
          YYERROR;
        }
      }
| VREG
      { $$ = vreg[$1]; }
| vexp '+' vexp
      { $$,hi = $1,hi + $3,hi;
        $$,lo = $1,lo + $3,lo; }
| dexp '+' vexp
      { $$,hi = $1 + $3,hi;
        $$,lo = $1 + $3,lo; }
| vexp '-' vexp
      { $$,hi = $1,hi - $3,lo;
        $$,lo = $1,lo - $3,hi; }
| dexp '-' vexp
      { $$,hi = $1 - $3,lo;
        $$,lo = $1 - $3,hi; }
| vexp '*' vexp
      { $$ = vmul( $1,lo, $1,hi, $3 ); }
| dexp '*' vexp
      { $$ = vmul( $1, $1, $3 ); }

```

```

|      vexp  '/'  vexp
|      {      if( dcheck( $3 ) ) YYERROR;
|              $$ = vdiv( $1.lo, $1.hi, $3 ); }
|
|      dexp  '/'  vexp
|      {      if( dcheck( $3 ) ) YYERROR;
|              $$ = vdiv( $1, $1, $3 ); }
|
|      '-'  vexp
|      {      Zprec UMINUS
|              $$hi = -$2.lo;  $$lo = -$2.hi; }
|
|      '('  vexp  ')'
|      {      $$ = $2; }
|
|

```

%%

```
# define BSZ 50      /* buffer size for floating point numbers */
```

```
/* lexical analysis */
```

```

yylex(){
    register ci

    while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

    if( isupper( c ) ){
        yyval.ival = c - 'A';
        return( VREG );
    }
    if( islower( c ) ){
        yyval.ival = c - 'a';
        return( DREG );
    }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if( isdigit( c ) ) continue;
            if( c == '.' ){
                if( dot++ != exp ) return( ',' );
                /* will cause syntax error */
                continue;
            }
        }
    }
}

```

```

        if( c == 'e' ){
            if( exp++ ) return( 'e' );
            /* will cause syntax error */
            continue;
        }

        /* end of number */
        break;
    }
    *cp = '\0';
    if((cp-buf) >= BSZ) printf("constant too long; truncated\n");
    else ungetc( c, stdin ); /* push back last char read */
    yylval.dval = atof( buf );
    return( CONST );
}
return( c );
}

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a; v.lo = b; }
    else { v.hi = b; v.lo = a; }

    if( c>d ) {
        if( c>v.hi ) v.hi = c;
        if( d<v.lo ) v.lo = d;
    }
    else {
        if( d>v.hi ) v.hi = d;
        if( c<v.lo ) v.lo = c;
    }
    return( v );
}

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
}

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return( 1 );
    }
    return( 0 );
}

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
}

```

Appendix D: Old Features Supported but Not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes “”.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or `_`, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where `%` is legal, backslash “\” may be used. In particular, `\ \` is the same as `%%`, `\left` the same as `%left`, etc.
4. There are a number of other synonyms:

```
Z< is the same as Zleft
Z> is the same as Zright
Zbinary and Z2 are the same as Znonassoc
Z0 and Zterm are the same as Ztoken
Z= is the same as Zprec
```

5. Actions may also have the form

```
=( . . . )
```

and the curly braces can be dropped if the action is a single C statement.

6. C code between `{` and `}` used to be permitted at the head of the rules section, as well as in the declaration section.

Table of Contents

The ADB Debugger

Introduction	1
Invocation	1
Command Format	2
Displaying Information	3
Debugging C Programs	5
Debugging A Core Image	5
Setting Breakpoints	7
Advanced Breakpoint Usage	11
Other Breakpoint Facilities	13
Maps	14
Variables and Registers	15
Formatted dumps	16
Patching	19
Anomalies	20
Command Summary	20
Formatted Printing	20
Breakpoint and Program Control	20
Miscellaneous Printing	20
Calling the Shell	20
Assignment to Variables	20
Format Summary	21
Expression Summary	21
Expression Components	21
Dyadic Operators	21
Monadic Operators	21

The ADB Debugger

Introduction

ADB is a debugging program that is available on HP-UX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB.

Invocation

ADB is invoked as:

```
adb objfile corefile
```

where `objfile` is an executable HP-UX file and `corefile` is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are `a.out` and `core` respectively. The filename minus (-) means "ignore this argument," as in:

```
adb - core
```

The `objfile` can be written to if `adb` is invoked with the `-w` flag as in:

```
adb -w a.out -
```

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request `!q` or `!Q` (or `CTRL-D`) must be used to exit from ADB.

Command Format

The general form of a request is:

[address] [,count] [command] [modifier]

ADB maintains a current address, called *dot*, similar in function to the current pointer in the HP-UX editor. When **address** is entered, dot is set to that location. The command is then executed count times.

Address and count are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type `name` or `_name`; ADB will recognize both forms. The default base for integer input is initialized to hexadecimal, but can be changed.

The following table illustrates some general ADB commands and meanings:

?	Print contents from <code>a.o u t</code> file
/	Print contents from <code>c o r e</code> file
=	Print value of "dot"
:	Breakpoint control
\$	Miscellaneous requests
;	Request separator
!	Escape to shell

A `CTRL-C` will terminate the execution of any command in ADB.

Displaying Information

ADB has requests for examining locations in either **objfile** or **corefile**. The `?` request examines the contents of **objfile**, the `/` request examines the **corefile**.

Following the `?` or `/` command the user specifies a format.

The following are some commonly used format letters:

<code>c</code>	one byte as a character
<code>x</code>	two bytes in hexadecimal
<code>X</code>	four bytes in hexadecimal
<code>d</code>	two bytes in decimal
<code>F</code>	eight bytes in double floating point
<code>i</code>	MC68000 instruction
<code>s</code>	a null terminated character string
<code>a</code>	print in symbolic form
<code>n</code>	print a newline
<code>r</code>	print a blank space
<code>^</code>	backup dot

A command to print the first hexadecimal element of an array of long integers named `ints` in C would look like:

```
ints/X
```

This instruction would set the value of `dot` to the symbol table value of `_ints`. It would also set the value of the dot increment to four. The dot increment is the number of bytes printed by the format.

Let us say that we wanted to print the first four bytes as a hexadecimal number and the next four as a decimal one. We could do this by:

```
ints/XD
```

In this case, `dot` would still be set to `_ints` and the dot increment would be eight bytes. The dot increment is the value which is used by the `newline` command. `newline` is a special command which repeats the previous command. It does not always have meaning. In this context, it means to repeat the previous command using a count of one and an address of `dot plus dot increment`. In this case, `newline` would set `dot` to `ints+0x8` and type the two long integers it found there, the first in hex and the second in decimal. The `newline` command can be repeated as often as desired and this can be used to scroll through sections of memory.

Using the above example to illustrate another point, let us say that we wanted to print the first four bytes in long hex format and the next four bytes in byte hex format. We could do this by:

```
ints/X4b
```

Any format character can be preceded by a decimal repeat character.

The count field can be used to repeat the entire format as many times as desired. In order to print three lines using the above format we would type:

```
ints,3/X4bn
```

The `n` on the end of the format is used to output a carriage return and make the output much easier to read.

In this case the value of `dot` will not be `_ints`. It will rather be `_ints+0x10`. Each time the format was re-executed `dot` would have been set to `dot` plus `dot` increment. Thus the value of `dot` would be the value that `dot` had at the beginning of the last execution of the format. `Dot` increment would be the size of the format: eight bytes. A `newline` command at this time would set `dot` to `ints+0x18` and print only one repetition of the format, since the count would have been reset to one.

In order to see what the value of `dot` is at this point the command:

```
,=a
```

could be typed. `=` is a command which can be used to print the value of **address** in any format. It is also possible to use this command to convert from one base to another:

```
0x32=oxd
```

This will print the value `0x32` in octal, hexadecimal and decimal.

Complicated formats are remembered by ADB. One format is remembered for each of the `?`, `/` and `=` commands. This means that it is possible to type:

```
0x64=
```

and have the value `0x64` printed out in octal, hex and decimal. And after that, type:

```
ints/
```

and have ADB print out four bytes in long hex format and four bytes in byte hex format.

To an observant individual it might seem that the two commands:

```
main,10?i
```

and

```
main?10i
```

would be the same.

There are two differences. The first is that the numbers are in a different base. The repeat factor can only be a decimal constant, while the count can be an expression and is therefore, by default, in a hex base.

The second difference is that a `newline` after the first command would print one line, while a `newline` after the second command would print another ten lines.

Debugging C Programs

Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate some of the useful information that can be gotten from a core file. The object of the program is to calculate the square of the variable `ival` by calling the function `sqr` with the address of the integer. The error is that the value of the integer is being passed rather than the address of the integer. Executing the program produces a core file because of a bus error.

Figure 1: C program with pointer bug

```
int ints[]=    {1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0};

int ival;
main()
{
    register int i;
    for(i=0;i<10;i++)
    {
        ival = ints[i];
        sqr(ival);
        printf("sqr of %d is %d\n",ints[i],ival);
    }
}

sqr(x)
int *xi
{
    *x **= *xi;
}
```

ADB is invoked by:

```
adb
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. This request can be used to check the validity of the parameters passed. As shown in Figure 2 we can see that the value passed on the stack to the routine `sqr` is a 1, which is not what we are expecting.

Figure 2: ADB output for program of Figure 1

```

$c
_main+0x30:  _sqr      (0x1)
--start+0x38:  _main     (0x1, 0xFFDD0)
$r
Ps         0x4
Pc         0x20CA  _sqr+0x14:  move.l    0x8(a6),-(a7)

d0         0x4900          a0         0x1
d1         0x800          a1         0xFFDD0
d2         0x0           a2         0x0
d3         0x0           a3         0x0
d4         0x0           a4         0x0
d5         0x0           a5         0x0
d6         0x0           a6         0xFFDAC
d7         0x0           sp         0xFFDAC
"sqr+e,5?ia"
_sqr+0xE:      move.l    0x8(a6),a0
_sqr+0x12:     move.l    (a0),-(a7)
_sqr+0x14:     move.l    0x8(a6),-(a7)
_sqr+0x18:     jsr      _almul
_sqr+0x1E:     addq.w   #0x8,a7
_sqr+0x20:
$e
_argc_value:  0x1
_errno:      0x0
_environ:    0xFFDDB
_argv_value: 0xFFDD0
_ints:      0x1
_ival:      0x1
__pfile:    0x0
__iob:      0x0
__ctype:    0x2020
__sobuf:    0x0
__lastbuf:  0x4E08
__sibuf:    0x0
tb_pwt4:    0x31D1411E
tb_pwt8:    0x30FC4501
tb_bcd:     0x10203
tb_pwt:     0x2F52FBAC
tb_auxpt:   0xACB0628
tb_pwt1:    0x32A50FFD
tb_bin:     0x10203
_end:       0x0
_edata:     0x

```

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location. The instruction printed for the pc does not always make sense. This is because the pc has been advanced and is either pointing at the next instruction, or is left at a point part way through the instruction that failed. In this case the pc points to the next instruction. In order to find the instruction that failed we could list the instructions and their offsets by the following command.

```
sqr+e,5?ia
```

This would show us that the instruction that failed was:

```
_sqr+0x12:move.l(a0),-(a7)
```

This is the first instruction before the value of the pc. The value printed out for register a0 also indicates that a dereference of its value would fail.

The request:

```
$e
```

prints out the values of all external variables at the time the program crashed.

Setting Breakpoints

Consider the C program in Figure 3. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

Figure 3: C program to decode tabs

```
#include <stdio.h>
#define MAXLINE 80
#define YES      1
#define NO       0
#define TABSP    8

char   input[] = "data";
FILE   *stream;
int    tabs[MAXLINE];
char   ibuf[BUFSIZ];

main()
{
    int col, *ptab;
    char c;

    setbuf(stdout, ibuf);
    ptab = tabs;
    settab(ptab); /*Set initial tab stops */
    col = 1;
    if((stream = fopen(input, "r")) == NULL) {
        printf("Zs : not found\n", input);
        exit(8);
    }
    while((c = getc(stream)) != EOF) {
        switch(c) {
            case '\t': /* TAB */
                while(tabpos(col) != YES) {
                    putchar(' '); /* put BLANK */
                    col++;
                }
                break;
            case '\n': /*NEWLINE */
                putchar('\n');
                col = 1;
                break;
            default:
                putchar(c);
                col++;
        }
    }
}
```

```

/* TabPos return YES if col is a tab stop */
tabPos(col)
int col;
{
    if(col > MAXLINE)
        return(YES);
    else
        return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
    int ii;

    for(i = 0; i<= MAXLINE; i++)
        (i>TABSP ? (tabs[i] = NO) : (tabs[i] = YES));
}

```

We will run this program under the control of ADB (see Figure 4) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```
settab+e:b
fopen+e:b
tabpos+e:b
```

set breakpoints at the starts of these functions. The above addresses are entered as `symbol+e` so that they will appear in any C backtrace since the first three instructions of each function is a standard sequence that links in the new function. Note that one of the functions is from the C library.

Figure 4: ADB output for C program of Figure 3

```

adb a.out -
executable file = a.out
ready
settab+e:b
fopen+e:b
tabpos+e:b
$b
breakpoints
count  bkpt      command
0x1    _tabPos+0xE
0x1    _fopen+0xE
0x1    _settab+0xE
:r
Process 11640 created
a.out: running
breakpoint  _settab+0xE:  clr.l  -0x4(a6)
settab+e:d
:c

```

```

a.out: running
breakpoint      _fopen+0xE:      jsr      __findiop
$c
_main+0x4B:     _fopen  (0x4E3B, 0x4E3E)
--start+0x2C:  _main   (0x1, 0xFFFFDE0)
_tabs/24X
_tabs:          0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0
                0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0
                0x1      0x0      0x0      0x0      0x0
                0x0      0x0      0x0      0x0      0x0

```

```

:c
a.out: running
breakpoint      _tabpos+0xE:     cmp.l   #0x50,0x8(a6)
:s
a.out: running
stopped at      _tabpos+0x16:    ble.s  _tabpos+0x1C
<newline>
a.out: running
stopped at      _tabpos+0x1C:    move.l  0x8(a6),d0
<newline>
a.out: running
stopped at      _tabpos+0x20:    asl.l  #0x2,d0
<newline>
a.out: running
stopped at      _tabpos+0x22:    add.l  #0x58A4,d0
<newline>
a.out: running
stopped at      _tabpos+0x28:    move.l  d0,a0
<newline>
a.out: running
stopped at      _tabpos+0x2A:    move.l  (a0),d0

```

```

:d*
:c
a.out: running
Process terminated
settab+e:b settab,5?ia
tabpos+e,3:b ibuf/20c
:r
Process 3255 created
a.out: running
settab,5?ia
_settab:        link    a6,#0xFFFFFFFF
_settab+0x4:     tst.b  -0x10(a7)
_settab+0x8:     movem.l #<,-0x4(a6)
_settab+0xE:     clr.l  -0x4(a6)
_settab+0x12:    cmp.l  #0x50,-0x4(a6)
_settab+0x1A:   breakpoint  _settab+0xE:     clr.l  -0x4(a6)

```

```

:c
a.out: running
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
breakpoint     _tabpos+0xE:     cmp.l  #0x50,0x8(a6)
$a
Process 3255 killed

```


To print the location of breakpoints one types:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count-1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function `settab` we see that the breakpoint is set after the instruction to save the registers on the stack. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at `settab` with the addresses of each location displayed.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function `settab`, one types:

```
settab+e:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for `fofen`), ADB requests can be used to display the contents of memory. For example:

```
$c
```

to display a stack trace, or:

```
tabs,3/8X
```

to print three lines of 8 locations each from the array called `tabs`. The format `X` is used since integers are four bytes on the MC68000. By this time (at location `fofen`) in the C program, `settab` has been called and should have set a one in every eighth location of `tabs`.

Advanced Breakpoint Usage

When we continue the program with:

```
:c
```

we hit our first breakpoint at `tabpos` since there is a tab following the “This” word of the data. We can execute one instruction by:

```
:s
```

and can single step again by hitting “carriage return”. Doing this we can quickly single step through `tabpos` and get some confidence that it is working. We can look at twenty characters of the buffer of characters by typing:

```
>buf/20c
```

Several breakpoints of `tabpos` will occur until the program has changed the tab into equivalent blanks. Since we feel that `tabpos` is working, we can remove all the breakpoints by:

```
:d*
```

If the program is continued with:

```
:c
```

it resumes normal execution and continues to completion after ADB prints the message:

```
a.out: running
```

It is possible to add a list of commands we wish to execute as part of a breakpoint. By way of example let us reset the breakpoint at `settab` and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+e:b settab,5?ia
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
tabpos+e,3:b ibuf/20c
```

This request will print twenty character from the buffer of characters at each occurrence of the breakpoint.

If we wished to print the buffer every time we passed the breakpoint without actually stopping there we could type:

```
tabpos+e,-1:b ibuf/20c
```

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+e:b settab,5?ia;ptab/o
```

could be entered after typing the above requests. The semicolon is used to separate multiple ADB requests on a single line.

Now the display of breakpoints:

```
$ b
```

shows the above request for the `settab` breakpoint. When the breakpoint at `settab` is encountered the ADB requests are executed.

Note

Setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+e:b .,5?ia  
fopen+e:b
```

will print the last thing dot was set to (in the example `fopen`) not the current location (`settab`) at which the program is executing.

The HP-UX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
: c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
: c 0
```

is typed.

Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile> outfile
```

This request kills any existing program under test and starts the `a.out` afresh. The process will run until a breakpoint is reached or until the program completes or crashes.

If it is desired to start the program without running it the command:

```
:e arg1 arg2 ... <infile> outfile
```

can be executed. This will start the process, and leave it stopped without executing the first instruction.

If the program is stopped at a subroutine call it is possible to step around the subroutine by:

```
:S
```

This sets a temporary breakpoint at the next instruction and continues. This may cause unexpected results if `:S` is executed at a branch instruction.

ADB allows a program to be entered at a specific address by typing:

```
address:r
```

The count field can be used to skip the first n breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first n breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by:

```
address:c
```

The program being debugged runs as a separate process and can be killed by:

```
:k
```

All of the breakpoints set so far can be deleted by:

```
:d*
```

A subroutine may be called by:

```
:x address [parameters]
```

Maps

HP-UX supports several executable file formats. These are used to tell the loader how to load the program file. A nonshared text program file is the most common and is generated by a C compiler invocation such as `cc prog.c`. A shared text file is produced by a C compiler command of the form `cc -n prog.c`, ADB interprets these different file formats and provides access to the different segments through the maps. To print the maps type:

```
$m
```

In nonshared files, both text (instructions) and data are intermixed. In shared files the instructions are separated from data and `?*` accesses the data part of the `a.out` file. The `?*` request tells ADB to use the second part of the map in the `a.out` file. Accessing data in the `core` file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. Figure 5 shows the display of two maps for the same program linked as a nonshared and shared respectively. The `b`, `e`, and `f` fields are used by ADB to map addresses into file addresses. The `f1` field is the length of the header at the beginning of the file (0x40 bytes for an `a.out` file and 0x800 bytes for a `core` file). The `f2` field is the displacement from the beginning of the file to the data. For a nonshared file with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

Figure 5: ADB output for maps


```
adb a.out.unshared core.unshared
$m
executable file = a.out.unshared
core file = core.unshared
ready
? map `a.out.unshared'
b1 = 0x2000    e1 = 0x20FC    f1 = 0x40
b2 = 0x2000    e2 = 0x20FC    f2 = 0x40
/ map `core.unshared'
b1 = 0x2000    e1 = 0x2400    f1 = 0x800
b2 = 0xFFF400  e2 = 0x1000000 f2 = 0xC00
$u
variables
b = 0x2000
d = 0x400
e = 0x2000
m = 0x107
s = 0xC00
$q
```

```
adb a.out.shared core.shared
$m
executable file = a.out.shared
core file = core.shared
ready
? map `a.out.shared'
b1 = 0x2000    e1 = 0x20FC    f1 = 0x40
b2 = 0x80000   e2 = 0x80000   f2 = 0x13C
/ map `core.shared'
b1 = 0x2400    e1 = 0x2800    f1 = 0x800
b2 = 0xFFF400  e2 = 0x1000000 f2 = 0xC00
$u
```

```

variables
b = 0x2400
d = 0x400
e = 0x2000
m = 0x108
s = 0xC00
t = 0x400
$#

```

 The `b` and `e` fields are the starting and ending locations for a segment. Given an address, `A`, the location in the file (either `a.o` or `core`) is calculated as:

$b1 \leq A \leq e1 \rightarrow \text{file address} = (A - b1) + f1$
 $b2 \leq A \leq e2 \rightarrow \text{file address} = (A - b2) + f2$

Variables and Registers


ADB provides a set of variables which are available to the user. A variable is composed of a single letter or digit. It can be set by a command such as:

```
0x32>5
```

which sets the variable 5 to hex 32. It can be used by a command such as:

```
<5=X
```

which will print the value of the variable 5 in hex format.


 Some of these variables are set by ADB itself. These variables are:

- `o` last value printed
- `b` base address of data segment
- `d` length of the data segment
- `e` The entry point
- `m` execution type (0x107 (nonshared), 0x108 (shared))
- `s` length of the stack
- `t` length of the text

These variables are useful to know if the file under examination is an executable or `core` image file. ADB reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, then the header of the executable file is used instead.

Variables can be used for such purposes as counting the number of times a routine is called. Using the example of Figure 3, if we wished to count the number of times the routine `tabPos` is called we could do that by typing the sequence:

```
0>5 tabPos+e,-1:b <5+1>5 :r <5=d
```

 The first command will set the variable 5 to zero. The second command will set a breakpoint at `tabPos+e`. Since the count is -1 the process will never stop there but ADB will execute the breakpoint command every time the breakpoint is reached. This command will increment the value of the variable 5 by 1. The `:r` command will cause the process to run to termination. And the final command will print the value of the variable.

`$v` can be used to print the values of all non-zero variables.

The values of individual registers can be set and used in the same way as variables. The command:

```
0x32>d0
```

will set the value of the register `d0` to hex 32. The command:

```
<d0=X
```

will print the value of the register `d0` in hex format. The command `$r` will print the value of all the registers.

Formatted dumps

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are some examples.

The line:

```
<b , -1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

`<b` The base address of the data segment.

`<b , -1` Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format `4o4^8Cn` is broken down as follows:

`4o` Print 4 octal locations.

`4^` Backup the current address 4 locations (to the original start of the field).

`8C` Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as `@` followed by the corresponding character in the range 0140 to 0177. An `@` is printed as `@@`.

`n` Print a new line.

The request:

```
<b , <d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (`<d` provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, `dump`, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/Bona
```

The request `120$w` sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request `4095$s` increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request `=` can be used to print literal strings. Thus, headings are provided in this `dump` program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request `$v` prints all non-zero ADB variables. The request `0$s` sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/Bona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 7 shows the results of some formatting requests on the C program of Figure 6.

Figure 6: Simple C program for Illustrating Formatting and Patching

```

char    str1[] "This is a character string";
int     one    1;
int     number 456;
long    lnum   1234;
float   fpt    1.25;
char    str2[] "This is the second character string";
main()
{
    one = 2;
}

```

Figure 7: ADB output illustrating fancy formats

```

adb a.out.shared -
executable file = a.out.shared
ready
<b,-1?Bona
_ str1:          052150  064563  020151  071440  060440  061550  060562  060543

_ str1+0x10:     072145  071040  071564  071151  067147  0        0        01

_ number:
_ number:       0        0710   0        02322  037640  0        052150  064563

_ str2+0x4:      020151  071440  072150  062440  071545  061557  067144  020143

_ str2+0x14:     064141  071141  061564  062562  020163  072162  064556  063400
<b,20?4o4^8Cn
_ str1:          052150  064563  020151  071440  This is
060440  061550  060562  060543  a charac
072145  071040  071564  071151  ter stri
067147  0        0        01      ns@'@'@'@'@'@a

_ number:       0        0710   0        02322  @'@'@aH@'@'@dR

_ fpt:          037640  0        052150  064563  ? '@'@'This
020151  071440  072150  062440  is the
071545  061557  067144  020143  second c
064141  071141  061564  062562  haracter
020163  072162  064556  063400

address not found in a.out file
<b,20?4o4^8t8Cna
_ str1:          052150  064563  020151  071440  This is
_ str1+0x8:      060440  061550  060562  060543  a charac
_ str1+0x10:     072145  071040  071564  071151  ter stri
_ str1+0x18:     067147  0        0        01      ns@'@'@'@'@'@a
_ number:
_ number:       0        0710   0        02322  @'@'@aH@'@'@dR
_ fpt:
_ fpt:          037640  0        052150  064563  ? '@'@'This
_ str2+0x4:      020151  071440  072150  062440  is the
_ str2+0xC:      071545  061557  067144  020143  second c
_ str2+0x14:     064141  071141  061564  062562  haracter
_ str2+0x1C:     020163  072162  064556  063400

address not found in a.out file
<b,a?2b8t^2cn
_ str1:          0x54    0x68          Th
0x69    0x73          is
0x20    0x69          i
0x73    0x20          s
0x61    0x20          a
0x63    0x68          ch
0x61    0x72          ar
0x61    0x63          ac
0x74    0x65          te
0x72    0x20          r
$#

```

Patching

Patching files with ADB is accomplished with the `write`, `w` or `W`, request (which is not like the editor `write` command). This is often used in conjunction with the `locate`, `l` or `L` request. In general, the request syntax for `l` and `w` are similar as follows:

```
?l value
```

The request `l` is used to match on two bytes, `L` is used for four bytes. The request `w` is used to write two bytes, whereas `W` writes four bytes. The `value` field in either `locate` or `write` requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, `file1` is created if necessary and opened for both reading and writing. `file2` can be opened for reading but not for writing.

For example, consider the C program shown in Figure 6. We can change the word “This” to “The” in the executable file for this program, `ex7`, by using the following requests:

```
adb -w ex7 -  
?l 'Th'  
?W 'The '
```

The request `?l` starts at dot and stops at the first match of “Th” having set dot to the address of the location found. Note the use of `?` to write to the `a.out` file. The form `?*` would have been used for a shared text file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of “Th” and print the entire string. Execution of this ADB request will set dot to the address of the “Th” characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -  
:e arg1 arg2  
flag/w 1  
:c
```

The `:e` request is used to start `a.out` as a subprocess with arguments `arg1` and `arg2`. If there is a subprocess running ADB writes to it rather than to the file so the `w` request causes `flag` to be changed in the memory of the subprocess.

Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the `link` instruction. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. If a `:s` command is executed at a branch instruction, and the branch is taken, the command will act as a `:c` command. This is because a breakpoint is set at the next instruction and if it is not reached, the process will not stop.

Command Summary

Formatted Printing

<code>? format</code>	print from <code>a.out</code> file according to <i>format</i>
<code>/format</code>	print from <code>core</code> file according to <i>format</i>
<code>= format</code>	print the value of <code>dot</code>
<code>?w expression</code>	write <i>expression</i> into <code>a.out</code> file
<code>/w expression</code>	write <i>expression</i> into <code>core</code> file
<code>?l expression</code>	locate <i>expression</i> in <code>a.out</code> file

Breakpoint and Program Control

<code>:b</code>	set breakpoint at <code>dot</code>
<code>:c</code>	continue running program
<code>:d</code>	delete breakpoint
<code>:k</code>	kill the program being debugged
<code>:r</code>	run <code>a.out</code> file under ADB control
<code>:s</code>	single step

Miscellaneous Printing

<code>\$b</code>	print current breakpoints
<code>\$c</code>	C stack trace
<code>\$e</code>	external variables
<code>\$f</code>	floating registers
<code>\$m</code>	print ADB segment maps
<code>\$q</code>	exit from ADB
<code>\$r</code>	general registers
<code>\$s</code>	set offset for symbol match
<code>\$v</code>	print ADB variables
<code>\$w</code>	set output line width


Calling the Shell

<code>!</code>	call <i>shell</i> to read rest of line
----------------	--

Assignment to Variables

<code>>name</code>	assign <code>dot</code> to variable or register <i>name</i>
-----------------------	---


Format Summary



a	the value of dot
b	one byte in hexadecimal
c	one byte as a character
d	two bytes in decimal
f	four bytes in floating point
i	MC68000 instruction
o	two bytes in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next <i>n</i> space tab
u	two bytes as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

Expression Components




decimal integer	e.g. 0d256
octal integer	e.g. 0277
hexadecimal	e.g. 0xff
symbols	e.g. flag _main
variables	e.g. <b
registers	e.g. <pc <d0
(expression)	expression grouping

Dyadic Operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise <i>and</i>
 	bitwise <i>or</i>
#	round up to the next multiple

Monadic Operators



~	not
*	contents of location
-	integer negate

Table of Contents

Getting Started

Introduction	1
Manual Organization	2
Conventions Used In This Manual	3
Using Other HP-UX Manuals	4
Overview of cdb	4
Overview of Interprocess Debugging	5
Compiling Programs	5
Conventions	7
Notational Conventions	7
Variable Name Conventions	8
Expression Conventions	10
Procedure Call Conventions	13
Running cdb	14
Example Program	15

Viewing Commands

File Code Viewing Commands	17
Print Current File, Procedure and Line Number	17
Change Files and Print First Executable Line	18
Print Groups of Lines	18
Print Window of Text	19
Move Forward/Backward from Current Line	20
Miscellaneous File Viewing Commands	21
Stack Viewing Commands	22
Trace Stack for Expr Levels	22
Set Viewing Location	23
Data Viewing Commands	24
Print Variable's Value	24
View Non-current Location Variables	25
List Command	26
Miscellaneous Data Viewing Commands	26
Display Formats	27

Job Control Commands

Run/Terminate the Program	29
Terminate Current Child Process	30
Continue After Breakpoint/Signal	31
Single Step After Breakpoint	32

Breakpoint Commands

Set a Breakpoint	36
List Breakpoints	38
Delete Breakpoints	39
Miscellaneous Breakpoint Commands	40

Assertion Control Commands and Signal Handling Commands

Assertion Control Commands	43
Create New Assertion	43
Modify an Assertion	44
Tracing Program Execution	45
Toggle the State	46
Delete All Assertions	46
Signal Handling Commands	47
Reverse Handling of Signal	48

Record, Playback and Other cdb Commands

Record and Playback Commands	51
Miscellaneous Record and Playback Commands	54
Other Commands	55

Index



Getting Started

Introduction

If you're like most people, reading computer manuals is not your favorite pastime. We strongly urge you to read the remainder of this chapter. This manual assumes that you have read these first few pages; if you choose not to do so, you are on your own.

One other note: the best way for us to improve the quality of documentation is through your feedback. Please use one of the reply cards at the back of this manual to tell us what was helpful, what was not, and why. Feel free to comment on depth, technical accuracy, organization, and style. Your comments are appreciated.

Manual Organization

Chapter 1: Getting Started

Explains the conventions used in the manual and identifies other manuals referenced within this one. This chapter then presents an overview of *cdb*, how to compile and execute programs with *cdb*, conventions of *cdb*, and example programs used in this tutorial.

Chapter 2: Viewing Commands

This chapter contains viewing commands: file, stack, data. Viewing commands allow you to look at code, procedure calling sequences, or the value of variables. There is also a section on the display formats used with the Data Viewing Commands.

Chapter 3: Job Control Commands

Describes job control commands which let you execute or terminate the program, as well as, continue or single-step after a breakpoint.

Chapter 4: Breakpoint Commands

Covers the breakpoint commands that are used to stop a program at a user specified location. The execution commands, that can be specified when that breakpoint is encountered, are also covered.

Chapter 5: Assertion Control Commands and Signal Handling Commands

Contains the assertion control commands that check user specified conditions after every statement. It also includes the signal handling commands which give you the ability to alter how the debugger deals with signals.

Chapter 6: Record, Playback and Miscellaneous Commands

Discusses the record and playback commands used to recreate automatically a program state and miscellaneous *cdb* commands which do not really fit into any other category.

Conventions Used In This Manual

The following naming conventions are used throughout this manual.

- **Italics** indicate files and HP-UX commands, system calls, and subroutines found in the *HP-UX Reference* manual as well as titles of manuals. Italics are also used for symbolic items either typed by you or displayed by the system as discussed below. Examples include */usr/lib/nls/american/prog.cat*, *date(1)*, and *pty(4)*. The parenthetic number shown for commands, system calls, and other items found in the *HP-UX Reference* is a convention used in that manual.
- **Boldface** is used when a word is first defined and for **general emphasis**.
- **Computer font** indicates a literal typed by you or displayed by the system. A typical example is:

```
cdb main.c
```

Note that when a command or file name is part of a literal, it is shown in computer font and not italics. However, if the command or file name is symbolic (but not literal), it is shown in italics as the following example illustrates:

```
cdb executable_file
```

In this case you would type in your own *executable_file*. If the command has optional arguments, they are designated by square brackets, [], as the example below shows:

```
[line]p[count]
```

- Unless otherwise stated, all references such as “see the *ptrace(2)* entry for more details” refer to entries in the *HP-UX Reference* manual. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* manual’s *Permuted Index*.

Using Other HP-UX Manuals

This manual may be used in conjunction with other HP-UX documentation. References to the manual described next are included, where appropriate, in the text.

- The *HP-UX Reference* manual contains the syntactic and semantic details of all commands and application programs, system calls, subroutines, special files, file formats, miscellaneous facilities, and maintenance procedures available on the Series 200/500 HP-UX Operating System.

Overview of *cdb*

cdb is a symbolic source-level debugger that provides a controlled execution environment for C, FORTRAN, and Pascal programs. This tool can be used to debug C, FORTRAN, or Pascal programs without needing to know internals.

The scenario in which you use *cdb* is: if you have problems in your program, you recompile the program and then use *cdb* to assist in finding and correcting errors.

This tutorial describes the commands needed to use *cdb*. The tutorial provides a description of the commands and each command's syntax. There are programming examples in which the more important commands are used.

There are certain hardware dependencies, symbol table dependencies, diagnostics, warnings, and bugs associated with *cdb*. The authoritative reference on these items is the manual page *cdb(1)* in the *HP-UX Reference*. Everything else about *cdb* is detailed or implied in its pages, along with a quick reference of all the commands presented in this tutorial.

Overview of Interprocess Debugging

Both *cdb* and *adb* are **interprocess** debuggers. Interprocess debuggers run separately from the programs processes being debugged.

In HP-UX, *cdb* (and *adb*) interact with the program being debugged through *ptrace(2)*. This intrinsic allows a parent process read and write memory, and register locations in a child process, as well as causes the child process to machine-instruction step, continue (**free run**), and terminate.

The debugger *cdb* is the **parent process** and the program being debugged is the **child process**. In this document the terms **child process**, **target program**, **your program**, and **program being debugged** are synonymous.

Compiling Programs

The C, FORTRAN, and Pascal compilers emit debugging information when you compile with the *-g* option. This debug information is massaged by the assembler (Pascal and FORTRAN bypass the assembler) and the linked output ends up in the executable program file. *cdb* needs this information to be able to debug your program. If you want to use *cdb* on a particular procedure, it **must** be compiled with the *-g* option. You don't have to compile your entire program with *-g* (it's usually easier to do it that way), but as a minimum, the **main** procedure must be compiled with *-g* (otherwise *cdb* can't debug your program).

```
cc -g program
```

Compiling with *-g* increases the size of your executable file considerably (for example, compiling *cdb* source with *-g* leads to a 6x increase). However, the memory requirements will not change appreciably because the debug data is not loaded into memory.

The *objectfile* is the executable program file which has had one or more of its component modules compiled with debug option(s) (for example, *-g*) turned on. The *-g* option causes the linker to append */usr/lib/end.o* to your *objectfile*. This support module */usr/lib/end.o* must be included as the last object file in the list of those linked, except for libraries included with the *-l* option of *ld(1)*. The */usr/lib/end.o* subroutine contains buffer space used by *cdb* during command line procedure calls. An increase of 200 bytes in memory requirements is caused by compiling with *-g*. (Some systems automate this; see the *cdb(1)* "Hardware Dependencies" section.) The default for *objectfile* is *a.out*.

The *corefile* is a core image from a failed execution of *objectfile*. The default for *corefile* is *core*. (Note: the Series 500 does not support corefiles.)

The options available are:

- d *dir* names an alternate directory where source files are located. They are searched in the order given. If a source file is not found in any alternate directory, the current directory is searched last.
- r *file* names a *record* file which is invoked immediately (for overwrite, not for append). See the section below entitled "Record and Playback Commands" for a description of this feature.
- p *file* names a *playback* file which is invoked immediately. See the section below entitled "Record and Playback Commands" for a description of this feature.
- S *num* sets the size of the string cache to *num* bytes. The default *num* depends on the symbol table format used. The option is not available for all formats. The string cache holds data read from *objectfile*.

There can only be one *objectfile* and one *corefile* per debugging session (activation of the debugger). The program (*objectfile*) is not invoked as a child process until you give an appropriate command (see the "Job Control Commands" chapter). The same program may be restarted, as different child processes, many times during one debugging session.

This debugger is a complex, interactive tool with many synergistic and combinatorial features. What you can do with it is often limited only by your imagination. Remember, however, that the debugger is only a **window** into the world consisting mostly of the program being debugged and the system it runs on. If something puzzling happens, you may need to consult a manual which describes the program or the system, in order to understand the behavior.

Conventions

The debugger remembers the current file, current procedure, current line, and current data location. They are a function of what you have been viewing (not necessarily executing) most recently. Many commands use these current locations as defaults; many commands set them as a side effect. It is important to keep this in mind when deciding what a command does in any particular situation.

For example, if you stop in procedure *asub*, then view procedure *bsub*, then ask for the value of local variable *i*, the debugger assumes that the variable belongs to procedure *bsub*.

Notational Conventions

Most commands are of the form [*modifier*] *command-letter* [*options*]. Numeric modifiers before and after commands can be any numeric expression. They need not be just simple numbers. A blank is required before any numeric *option*. Multiple commands on one line must be separated by ;.

These are common modifiers and other special notations:

(A B C)	Any one of A or B or C is required.
[A B C]	Any one of A or B or C is optional.
<i>file</i>	A file name.
<i>proc</i>	A procedure (or function, or subroutine) name.
<i>var</i>	A variable name.
<i>number</i>	A specific, constant number (e.g. 9, not 4+5). Floating point (real) numbers may be used any place a constant is allowed.
<i>expr</i>	Any expression, but with limitations stated below.
<i>depth</i>	A stack depth, as printed by the <i>t</i> command. The top procedure is at a <i>depth</i> of zero. A negative <i>depth</i> acts like a <i>depth</i> of zero. Stack depth usually means exactly at the specified depth , not the first instance at or above the specified depth .

- format* A style for printing data. See the “Viewing Commands” chapter for details.
- commands* A series of debugger commands, separated by ;, entered on the command line, or saved with a breakpoint or assertion. Semicolons are ignored (as commands) so they can be freely used as command separators. Commands may be grouped with {} for the *a*, *b*, *if*, and *!* commands. In all other cases, commands inside {} are ignored.

Variable Name Conventions

Variables are referenced exactly as they are named in your source file(s). Case sensitivity is controlled by the *Z* command. Be careful with one letter variable names, since they can be confused with commands. If an expression begins with a variable that might be mistaken for a command, just enclose the expression in () (e.g. (*k*)), or eliminate any white space between the variable and the first operator (use *k= 9* instead of *k = 9*).

If you are interested in the value of some variable *var*, there are a number of ways of getting it, depending on where and what it is:

- var* Search the stack for the most recent instance of the current procedure. If found, see if *var* is a parameter or local variable of that procedure. If not, search outward using scoping rules for *var*.
- proc.var* Search the stack for the most recent instance of *proc*. If found, see if it has a parameter or local variable named *var*, as before.
- proc.depth.var* Use the instance of *proc* that is at depth *depth* (exactly), instead of the most recent instance. This is very useful for debugging recursive procedures where there are multiple instances on the stack.
- :var* Search for a global (not local) variable named *var*.

Dot is shorthand for the last thing you viewed (see the “Data Viewing Commands” section). It has the same size it did when you last viewed it. For example, if you look at a **long** as a **char**, then **.** is considered to be one byte long. This is useful for treating things in unconventional ways, such as changing the second highest byte of a **long** without changing the rest of the **long**. *Dot* may be treated like any other variable.

NOTE

The **.** (*dot*) is the **name** of this magic location. If you use it, it is de-referenced like any other name. If you want the **address** of something that is, say, 30 bytes farther on in memory, do not use **+.30**. That would take the contents of *dot* and add 30 to it. Instead, say **@+.30**, which adds 30 to the **address** of *dot*.

Special variables are names for things that are not normally directly accessible. Special variables include:

<i>\$var</i>	The debugger has room in its own address space for a number of user-created special variables. They are all of type long , and do not take on the type of any expression they are assigned to. Names are defined when they are first seen. For example, saying <i>\$xyz = 3*4</i> creates special symbol <i>\$xyz</i> , and assigns to it the value 12. Special variables may be used just like any other variables.
<i>\$pc, \$fp, \$sp, \$r0, etc.</i>	These are the names of the program counter, the frame pointer, the stack pointer, the registers, etc. To find out which names are available on your system, use the <i>l r</i> (list registers) command. All registers act as type integer .
<i>\$result</i>	This is used to reference the return value from the last procedure exit. Where possible, it takes on the type of the procedure. <i>\$short</i> and <i>\$long</i> are available as alternate ways of looking at <i>\$result</i> .
<i>\$signal</i>	This lets you see and modify the current child process signal number.

\$lang	This lets you see and modify the current language (0 for C, 1 for FORTRAN, or 2 for Pascal).
\$line	This lets you see and modify the current source line number, which can be set with a number of different commands.
\$malloc	This lets you see the current amount of memory (bytes) allocated at run-time for use by the debugger itself.
\$cBad	This lets you see and modify the number of machine instructions the debugger will step while in a non-debuggable procedure before setting an up-level breakpoint and free-running to it. Setting it to a small value can improve debugger performance, at the risk of taking off free-running after missing the up-level break for some reason.

To see all the special variables, including the predefined ones, use the *ls* (list specials) command.

You can also look up code addresses with:

proc#line

which searches for the given procedure name and line number (which must be an executable line within *proc*) and uses the code address of that line. Just referring to a procedure *proc* by name uses the code address of the entry point to that procedure.

Expression Conventions

Every expression has a value, even simple assignment statements, as in C. *Naked* expression values (those which aren't command modifiers) are always printed unless the next token is `;` (command separator) or `}` (command block terminator). Thus breakpoint and assertion commands (see the appropriate sections below) are normally silent. To force an expression result to be printed, follow the expression with */n* (print in normal format; see below).

Integer constants may begin with *0* for octal or *0x* or *0X* for hexadecimal. They are **int** if they fit in two bytes, **long** otherwise. If followed immediately by *l* or *L*, they are forced to be of type **long** (this is useful on systems where **int** is two bytes).

Floating point constants must be of the form:

digits.digits[e | E | d | D | L | l [+|-]*digits*]

for example, 1.0, 3.14e8, or 26.62D-31. One or more leading digits is required to avoid confusion with . (*dot*). A decimal point and one or more following digits is required to avoid confusion for some command formats. If the exponent doesn't exactly fit the pattern shown, it is not taken as part of the number, but as separate token(s). The *d* and *D* exponent forms are allowed for compatibility with FORTRAN. The *l* and *L* exponent forms are allowed for compatibility with Pascal. However, all floating point constants are taken as doubles, regardless.

Character constants must be entered in single quotes (for example, 'n') and are treated as **integers**. C string constants must be entered in double quotes (for example, "Hello World") and are treated like *char ** (i.e., pointer to **char**). FORTRAN and Pascal strings may be enclosed in either single quotes '' or double quotes "". Character and string constants may contain the standard backslashed escapes understood by the C compiler and the *echo(1)* command, including \b, \f, \n, \r, \t, \\, \', and \nnn. However, \Return is not supported, in quotes or at the end of a command line.

Expressions are composed of any combination of variables, constants, and C operators. If the debugger is invoked as *cdb*, the C operator *sizeof* is also available. If the debugger is invoked as *fdb*, FORTRAN operators are also available and FORTRAN meanings take precedence where there is a conflict. The same is true for Pascal if the debugger is invoked as *pdb*.

If there is no active child process and no *corefile*, you can only evaluate expressions containing constants.

Expressions approximately follow the C rules of promotion, e.g. **char**, **short**, and **int** become **long**, and **float** becomes **double**. If either operand is a **double**, floating point math is used. If either operand is **unsigned**, unsigned math is used. Otherwise, normal (integer) math is used. Results are then cast to proper destination types for assignments.

If a floating point number is used with an operator that doesn't normally permit it, the number is cast to **long** and used that way. For example, the C binary operator ~ (bit invert) applied to the constant 3.14159 is the same as ~3.

Note that = means **assign** except in Pascal. In Pascal, = is a comparison operator; use := for assignments. For FORTRAN use == or .EQ.. For example, if you invoke the debugger as *cdb*, then set \$lang = 2 (Pascal), you must say \$lang := 0 to return to C.

Use // for division, instead of /, to distinguish from display formatting (see the “Data Viewing Commands” section).

The special unary operator \$in (not to be confused with debugger local variables) evaluates to 1 (true) if the operand is an address inside a debuggable procedure and \$pc (the current child process program location) is also in that procedure, else it is 0 (false). For example, \$in main is true if the child process is stopped in *main()*.

If the first expression on a line begins with + or -, use () around it to distinguish from the + and - commands (see the “Data Viewing Commands” section). Parentheses may also be needed to distinguish an expression from a command it modifies.

You can attempt to dereference any constant, variable, or expression result using the C * operator. If the address is invalid, an error is given.

Whenever an array variable is referenced without giving all its subscripts, the result is the address of the lowest element referenced. For example, consider an array declared as x[5][6][7] in C, x(5,6,7) in FORTRAN, or x[1..5,2..6,3..7] in Pascal. Referencing it simply as x is the same as just x in C, the address of x(1,1,1) in FORTRAN, or the address of x[1,2,3] in Pascal. Referencing it as x[4] is the same as &(x[4][0][0]) in C, the address of x(1,1,4) in FORTRAN, or the address of x[4,2,3] in Pascal.

If a not-fully-qualified array reference appears on the left side of an assignment, the value of the right-hand expression is stored into the element at the address specified.

String constants are stored in a buffer in the file */usr/lib/end.o*. The debugger starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If the debugger reaches the end of the buffer, it goes back and reuses it from the beginning. In general this doesn't cause any problems. However, if you use very long strings, or if you assign a string constant to a global pointer, problems could arise.

Procedure Call Conventions

Procedures may be invoked from the command line, even within expressions. For example:

```
xyz = $abc * (3 + def (ghi - 1, jkl, "Hi Mom"))
```

calls procedure *def* when its value is needed in the expression.

Any breakpoints encountered during command line procedure invocation are handled as usual. However, the debugger has only one active command line at a time. If it stops in a called procedure for any reason, the remainder (if any) of the old command line is tossed, with notice given.

If you attempt to call a procedure when there is no active child process, one is started for you as if you gave a single-step command first. Unfortunately, this means that the data in *corefile* (if any) may disappear or be reinitialized.

If you send signal SIGINT (e.g., hit the **Break** key) while in a called procedure, the debugger aborts the procedure call and returns to the previous stopping point (the start of the main program for a new process).

You can call any procedure that is in your *objectfile*, even if it is not debuggable (was not compiled with the *-g* option). For example, assume that you reference *printf()* in your program, so the code for it is in your *objectfile*. Then you can enter on the command line:

```
printf ("This works! %d %c\n", 9, '?');
```

If you wonder what procedures are available, do a list labels command (*! l*). If you want to have some library routines available for debugging, but they aren't referenced anywhere in your code (so they aren't linked), relink your program with the *-u* option to force their inclusion.

Note that procedure name *_end_* is declared in *end.c*.

Running `cdb`

If an `a.out` file exists, then you invoke `cdb` by typing:

```
cdb
```

Otherwise, you need to specify an executable file as shown below.

To invoke the debugger on your C program, type:

```
cdb executable_file
```

Run the debugger on FORTRAN programs via:

```
fdb executable_file
```

and on Pascal programs via:

```
pdb executable_file
```

`/bin/fdb` and `/bin/pdb` are links to `/bin/cdb`. The `cdb` debugger does some language-dependent processing based on how it was invoked (`cdb`, `fdb`, or `pdb`). Examples of this are:

- FORTRAN arrays (column-major storage)
- FORTRAN `CHAR*` and Pascal `string` variables
- Pascal `PACKED arrays of CHAR`

You may change the **current** language from within `cdb/fdb/pdb` with the `$lang` special variable (see the “Conventions” section for more details).

Throughout the remainder of this document, `cdb` will be used as a generic term for `cdb/fdb/pdb`.

The `cdb` debugger needs to be able to access the source files for your program. The debugger assumes they are in the current directory. If they're not, use the `-d` command line option to specify their location. For example:

```
cdb -d src1 -d src2 bin/pgm
```

runs `cdb` on `./bin/pgm` with source in `./src1` and `./src2`.

The *cdb* debugger starts up by displaying file and procedure counts and then the first executable line of your program. At this point your program has **not** been loaded into memory.

cdb then prompts for commands with the > character.

Example Program

The example program used throughout this tutorial is listed below. Almost all the *cdb* commands covered in this tutorial will be illustrated using these two files (*main.c* and *sub.c*). Type them in exactly as shown, using an appropriate text editor (e.g., *vi*). Then compile them both and start *cdb*.

For the purposes of this document, the file *main.c* must contain the main program:

```
main ()
{
    long i;
    i = 5;
    asub(i);
}
```

The file *sub.c* must contain the subroutines:

```
asub (arg)
long arg;
{
    bsub(arg);
}

bsub (myarg)
long myarg;
{
    /* do nothing */
}
```

To compile these program files, use the C compiler and the *-g* option as shown in the previous section “Compiling Programs”. Type:

```
cc -g main.c sub.c
```

During compilation, two object files *main.o* and *sub.o* will be created and placed in the current directory. You can use the *lsf* command to check for them. To start the debugger on the **default executable object file** *a.out* type:

```
cdb a.out
```

NOTE

All examples in this tutorial were run on the Series 500, therefore addresses will differ from those on the Series 200.

Viewing Commands

A user can view the source code statically (before the program has executed) or dynamically (during execution). The stack and data, on the other hand, are meaningless until the program is executing and a breakpoint is reached.

File Code Viewing Commands

One must understand the concept of **current** lines, files, and procedures in order to use *cdb*. The *cdb* debugger interprets everything relative to the **current** viewing location; this holds particularly to line numbers and variable names.

Print Current File, Procedure and Line Number

Syntax:

```
e
```

Example:

```
$cdb a.out
Source files:    3
Procedures:     4
main.c: main: 4: i=5;
>e
main.c: main: 4: i=5;
```

This command prints the line you are presently located at within the file. It shows the current file, procedure, line number, and source line (*main.c: main: 4: i = 5;*). Commands that show the file and procedure with a source line skip (do not print) any leading white space from the source line.

Change Files and Print First Executable Line

Syntax:

```
e file
e proc
```

This command places you in the file or procedure designated. Entering a file sets the current line number to 1. Entering a procedure sets the current file and line to the first executable line of the procedure. You can enter **any** file and look at it from *cdb*; it does not have to be a program source file.

Example:

```
>e sub.c
sub.c: 1: asub (arg)
>e asub
sub.c: asub: 4: bsub (arg);
```

Notice that the second *e* command places you into the *sub.c* file at the first executable line of *asub()*. To return to *main.c* simply type:

```
>e main.c
main.c: 1: main()
```

Print Groups of Lines

Syntax:

```
[line]p [count]
```

The *p* command can be used several ways. When *p* is used alone, the current line is output. Using *p* with just *line* prints the line specified by that number. If a *count* follows the *p*, *count* lines will be printed starting at *line*. *p* followed only by *count*, prints from the current line forward *count* lines. If more than one line is printed, the current line is marked with a = in the leftmost position.

Example:

```
>p
1: main ()
>5p
5:      asub(i);
>p2
5:      asub(i);
= 6:  }
>2p 3
2:  {
3:      long i;
= 4:      i = 5;
```

Print Window of Text

Syntax:

```
[line] w [window size]
[line] W [window size]
```

Instead of using *p* to print sections of text, sometimes the *w* and *W* commands are more useful. The window commands are used for quickly scrolling through source files (or any file). These commands print blocks of text thereby reducing the need to refer to paper listings during a debugging session. Window commands (*w* defaults to 11 lines and *W* defaults to 21 lines) print the block of text centered around the current line (or any specified line). The *line* parameter specifies the current line number. Then *window size* designates how many lines around the current line are printed.

You can cause the previous *w* or *W* command to be repeated by pressing **Return**. This causes the next successive block of text to be displayed. The *cdb* debugger remembers the size and direction of text windowing for the next **Return** command.

Example:

```
>e sub.c
sub.c: 1: asub (arg)
>5 p
5:  }
>w
1:  asub (arg)
2:  long arg;
3:  {
4:      bsub(arg);
= 5:  }
6:
7:  bsub (myarg)
8:  long myarg;
9:  {
10:     /* do nothing */
11:  }
>4 w
1:  asub (arg)
2:  long arg;
3:  {
= 4:      bsub(arg);
5:  }
6:
7:  bsub (myarg)
8:  long myarg;
9:  {
10:     /* do nothing */
11:  }
```

```

>w 5
  2: long arg;
  3: {
=  4:     bsub(arg);
  5: }
  6:
>9 w 5
  7: bsub (myarg)
  8: long myarg;
=  9: {
 10:     /* do nothing */
 11: }

```

Move Forward/Backward from Current Line

Syntax:

```

+[lines]
-[lines]

```

This command moves the cursor *lines* forward when you use *+* and *lines* backward when you use the *-*. The default is 1.

Example:

```

>- 3
  6:
>+ 4
 10:     /*do nothing*/

```

The window command and these directional commands can be blended to build the *+/-W/w* commands which are useful for changing direction. The *-W* and *-w* commands cause the preceding block of text to be displayed. While *+W* and *+w* cause the following block of text to be displayed.

Miscellaneous File Viewing Commands

- dir directory* Add *directory* to the list of alternate source directories. The effect is the same as using the *-d* invocation option. If the file containing the main procedure does not reside in the current directory, its directory must be specified with the *-d* option.
- L* This is a synonym for *OE* (see the “Set Viewing Location” section”).
- line* Print source line number *line* in the current file.
- +w[size]*
+W[size] Print a window of text, of the given or default *size*, beginning at the end of the previous window, if the previous command was a window command, or at the current line otherwise.
- w[size]*
-W[size] Print a window of text, of the given or default *size*, ending at the beginning of the previous window, if the previous command was a window command, otherwise end at the current line.

If after any window command you give a *w* or *W* command with no *line* specified, the debugger prints the following window of source text; or if the previous window command was *-w* or *-W* the previous window is printed, using the given *size* (or the default if none). Pressing **Return** after any window command does the same thing, but uses the previous *size* as well.

/[string] Search forward through the current file, from the line after the current line, for *string*.

?[string] Search backward for *string*, from the line before the current line.

Searches wrap around the end or beginning of the file, respectively. If *string* is not specified, the previous one is used. **Wild cards** and regular expressions are not supported; *string* must be literal. **Case sensitivity** is controlled by *z*; the default is **insensitive** (see the section “Other Commands” for details).

n Repeat the previous */* or *?* command using the same *string* as previously.

N The same as *n*, but the search goes in the opposite direction as specified by the previous */* or *?* command.

These search commands, */*, *?*, *n*, and *N* work the same as in *vi(1)*.

Stack Viewing Commands

These commands are only meaningful after the child process stops (e.g. on a breakpoint) because there is nothing on the stack until the child process is running. The procedure calling chain is displayed with the *t* and *T* commands.

A detailed description for using and setting breakpoints is provided in the “Breakpoint Commands” section. For this example type:

```
>b                               (set the breakpoint)
Added:
  1: count: 1  asub: 4: bsub(arg);
>r                               (run the program)
Starting process 1246: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
```

Trace Stack for Expr Levels

Syntax:

```
[depth] t
[depth] T
```

The *t* command traces the stack for the first *depth* (default 20) level and displays the procedures on the stack and their parameter values. The *T* supplements this information with local variables which are also displayed, using the */n* format (except that arrays and pointers are shown as addresses, and only the first word of structures is shown).

Example:

```
>t
0 asub (arg = 5)      [sub.c: 4]
1 main ()           [main.c: 5]
2 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
3 unknown ()
>T
0 asub (arg = 5)      [sub.c: 4]
1 main ()           [main.c: 5]
  i                 = 5
2 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
3 unknown ()
```

Non-debuggable procedures are also displayed but their parameters are displayed in hexadecimal.

Set Viewing Location

Syntax:

`[depth]E`

The *E* command sets the current viewing location to the procedure on the stack at depth *depth* and prints the current file name, procedure name, and line. The point of suspended execution is at *depth* = 0. For example, with the above stack trace the command `1E` sets the current viewing line to line 5 in *main.c* which is the call to *asub()*.

The *E* command only sets the **viewing** location. This means that using *E* to set the location to a prior instance of a recursive procedure and then querying the value of variable *x* will show *x* in the most recent instance of the procedure. The *proc.depth.var* syntax must be used in this case.

The *E* command is handy for quickly looking at the source code for the calling chain (perhaps to determine the context of the current procedure call). You use *OE* or its synonym *L* to get back to the point of suspended execution after roaming around setting breakpoints or viewing other files, etc.

Example:

```
>E
sub.c: asub: 4: bsub(arg);
>1E
main.c: main: 5 +0x0000000c: asub(i);
>OE
sub.c: asub: 4: bsub(arg);
```

Data Viewing Commands

Print Variable's Value

Syntax:

```
expr  
expr/format  
expr?format
```


The *expr* can be as simple as the name of a variable in a child process; or it can be a complex combination of variables and arithmetic operators. See the "Expression Conventions" section for further discussion. The debugger returns the value of the variable designated by *expr*. It is handled as if you had typed *expr/n* (print expression in normal format), unless followed by *;* or *}*, in which case nothing is printed.

All the variables in *expr* must be known in the current viewing location. For example, if you try to query the value of *arg* when the current location is not in *asub()*, you will receive the error message *Unknown name or command "arg"*.

If there is a conflict between a variable name and a command, the command name takes precedence. To query the value of such a variable, either enclose the name in parentheses, or specify a format. For example, *i* in *main()* conflicts with the *if* command:

Example:

```
>arg  
arg = 5  
>e main.c  
main.c: 1: main()  
>i  
Missing "{"  
>(i)  
i = 5
```




Sometimes during debugging it is necessary to print the contents of a variable using a different *format* than the normal default format (*n*). In the example below *i* is printed out in decimal as an integer. There are a variety of *formats* available (see “Miscellaneous Data Viewing Commands” and “Display Formats”). The */* specifies printing the value of the *expr* and the *?* designates printing the address of the *expr*. Then *^* indicates backing up to the preceding location while the *.* reverses the direction again to forward.

```
>i/d
i = 5
>i?d
-1073741424
>~/d
0xc000018c 56
>./d
0xc000018c 56
```

View Non-current Location Variables

Syntax:

```
proc.var
proc.depth.expr
```



With these forms you can view variables in a procedure not containing the current viewing location or look at a variable at a particular depth on the procedure stack (useful for recursive programs).

Example:

```
>asub.arg
arg = 5
>asub.1.arg
Procedure "asub" not found at stack depth 1
>main.1.i
i = 5
```


List Command

Syntax:

```
l[proc [depth]]
l (a | b | d | z)
l (f | g | l | p | r | s) [string]
```

This command *l* lists all parameters and local variables of the current procedure or the specified *proc* (if given) at the specified *depth* (if any). Data is displayed using */n* format, except that all arrays and pointers are shown simply as addresses and only the first word of any structure is shown.

The letters in parentheses stand for assertions, breakpoints, directories (where to search for files), signals (signal actions), files (sourcefiles), global variables (known to linker), labels, procedure names, registers, or special variables. If *string* is present, only those things with the same initial characters are listed.

Example:

```
>l main
i          = 5
>l a
No assertions
>l b
1: count: 1  asub: 4: bsub(arg);
>l f
0:  main.c          0x60100000 to 0x60100019
1:  sub.c           0x60180000 to 0x60180023
2:  end.c           0x60200000 to 0x60200007
>l p as
1:  asub            0x60180000 to 0x60180015
```

Miscellaneous Data Viewing Commands

- expr/format* Print the contents (value) of *expr* using *format*. For example, *abc/x* prints the contents of *abc* as an **integer**, in hexadecimal.
- expr?format* Print the address of *expr* using *format*. For example, *abc?o* prints the address of *abc* in octal.
- ^[[/]*format** Back up to the preceding memory location (based on the size of the last thing displayed). Use *format* if supplied, or the previous *format* if not. Note that no */* is needed after the *^*. Also note that you can reverse direction again (e.g., start going forward) by entering *.* (*dot*), which is always an alias for the current location, followed by **Return**.

Display Formats

Display formats are used only with Data Viewing Commands. The *format* is of the form: `[*][count]formchar[size]`.

* means **use alternate address map** (e.g., *abc*), if maps are supported.

The *count* is the number of times to apply the format style *formchar*. It must be a **number** not an expression.

The *size* is the number of bytes to be formatted for each *count*, and overrides the default *size* for the format style. It must be a positive decimal *number* (except short hand notations, see below). The *size* is disallowed with those *formchar*'s where it makes no sense.

For example, `abc/4x2` prints, starting at the memory location of *abc*, four two-byte numbers in hexadecimal.

Using an optional upper-case letter with formats that print numbers has the same affect as appending the *l* option to the format (see below). For example, `O` prints 4 bytes in octal (i.e., **long**). These formats, which are useful on systems where **integer** is shorter than **long**, are noted below. The following formats are available:

- n* Print in the *normal* format, based on the type. Arrays of **char** and pointers to **char** are interpreted as strings, and structures are fully dumped.
- (*d D*) Print in decimal (as **integer** or **long**).
- (*u U*) Print in unsigned decimal (as **integer** or **long**).
- (*o O*) Print in octal (as **integer** or **long**).
- (*x X*) Print in hexadecimal (as **integer** or **long**).
- (*b B*) Print a byte in decimal (either way).
- (*c C*) Print a character (either way).
- (*e E*) Print in *e* floating point notation (as **float** or **double**) (see *printf* (3)). Remember that floating point constants are always doubles.
- (*f F*) Print in *f* floating point notation (as **float** or **double**).
- (*g G*) Print in *g* floating point notation (as **float** or **double**).
- a* Print a string using *expr* as the address of the first byte.

- s* Print a string using *expr* as the address of a pointer to the first byte. This is the same as saying **expr/a*, except for arrays.
- t* Show the type of *expr* (usually a variable or procedure name). For true procedure types you must actually call the procedure, (e.g., *def (2)/t*; alone *def* is the address of the function, i.e., an integer).
- p* Print the name of the procedure containing address *expr*.
- S* Do a formatted dump of a structure (only with symbol tables which support it). Note that *expr* must be the address of a structure, not the address of a pointer to a structure.

There are some shorthand notations for *size*:

- b* 1 byte (**char**).
- s* 2 bytes (**short**).
- l* 4 bytes (**long**).

These can be appended to *formchar* instead of a numeric *size*. For example, *abc/xb* prints one byte in hexadecimal.

If you view an object with a *size* (explicitly or implicitly) less than or equal to the size of a **long**, the debugger changes the basetype to something appropriate for that *size*. This is so *.* (*dot*) works correctly for assignments. For example, *abc/c2* sets the type of *.* to **short**. One side effect is that if you look at a **double** using a **float** format, *dot* loses accuracy or has the wrong value.

Job Control Commands

The parent (*cdb* debugger) and the child (*objectfile*) processes take turns running. The debugger is only active while the child process is stopped due to a signal, including hitting a breakpoint, or terminated for whatever reason.

Run/Terminate the Program

Syntax

```
R
r[arguments]
```

Use *R* to run a new child process with no *argument* list and *r* to run a new child process with a given *argument* list (or the previous list if none is given). The existing child process, if any, is terminated first.

The *r* command is the most versatile way to begin program execution. The *arguments* list can contain *<* and *>* for redirecting standard input and standard output. (*<* does an *open(2)* of file descriptor 0 for read-only; *>* does a *creat(2)* of file descriptor 1 with mode 0666). The *arguments* list may contain shell variables and metacharacters, quote marks, or other special syntax. Special shell syntax is expanded by a Bourne shell. Because *{}* are shell metacharacters, *r* cannot be safely saved in a breakpoint or assertion command list.

If no *arguments* are given, the ones used with the last *r* command are used again. No arguments are used if *R* was used last. For example, the command line:

```
>r arg1 arg2 arg3 >file1 <file2
```

passes *arg1*, *arg2*, and *arg3* as arguments and redirects *stdin* and *stdout*. It is equivalent to running your program from the shell as in:

```
program arg1 arg2 arg3 >file1 <file2
```

The *r* command expands shell variables and meta-characters before passing the argument string to the child process. Remember, it always kills off an existing child process first. You can do this manually with the *k* command, too (see example under the “Terminate” section).

Where the *r* command starts your program and lets it free run, the *R* command works similarly, except no arguments or I/O redirection can be specified.

Example:

```
>r
Starting process 942: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
>r arg1 arg2
Terminating process 942
Starting process 947: "a.out arg1 arg2"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
>R
Terminating process 947
Starting process 948: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
```

Whenever *cdb* stops and displays a line of your program, that line has not been executed yet. So setting a breakpoint (see the "Breakpoint Commands" section) on a line will cause *cdb* to stop before executing any code for the statement(s) on that line.

Terminate Current Child Process**Syntax:**

```
k
```

Terminate (kill) the current child process if one exists.

Example:

```
>k
Really terminate child process? y
Terminating process 948
```

Continue After Breakpoint/Signal

Syntax:

```
[count]c[line]  
[count]C[line]
```

The *c* command causes execution to continue after a breakpoint or signal, while ignoring the signal, if any. The *C* command allows the signal, if any, to be received. This is fatal to the child process if it does not catch or ignores the signal.

There are two fields associated with a breakpoint: *count* and *command*. The *count* field is discussed here; the *command* field is explained later in the “Breakpoint Commands” section. The *count* field associated with a breakpoint is the number of times the breakpoint is encountered prior to recognition. If the *count* is positive, the breakpoint is **permanent** and *count* decrements with each encounter. When *count* goes to zero, the breakpoint is recognized and the *count* is reset to one. If *count* is negative, the breakpoint is **temporary** and *count* increments with each encounter. Once *count* is zero, the breakpoint is recognized, then deleted.

NOTE

Count is set to -1 (temporary) or 1 (permanent) for any new breakpoint. Only then can it be modified by the continue (*c*) command.

The *line*, if given, designates a temporary breakpoint at that line number, with a count of -1.

Example:

```
>r  
Starting process 942: "a.out"  
  
breakpoint at 060180006  
sub.c: asub: 4: bsub(arg);  
>c 11          **temporary breakpoint**  
Added:  
  2: count: -1 (temporary) bsub: 11: }  
  
breakpoint at 0x60180022  
sub.c: bsub: 11: }  
>c  
Child process terminated normally  
>e main  
main.c: main: 4: i = 5;
```

```

>5
 5:      asub(i);
>b
Added:
 2: count: 1   main: 5: asub(i);
>r
Starting process 1029: "a.out"

breakpoint at 0x601000a
main.c: main: 5: asub(i);
>C

breakpoint at 0x60180006
sub.c:  asub: 4: bsub(arg);

```

Single Step After Breakpoint

Syntax:

```

[count]s
[count]S

```

If there is no child process currently active, you can **step** into your program with the *s* and *S* commands. These commands start your program and then stop before the first executable line of the main procedure.

With these two commands, you can execute your program a source line at a time. The *s* command traces debuggable procedure calls and enters the debuggable procedure. It single steps 1 (or *count*) statements. Successive **Return**'s repeat with a *count* of 1. If *count* is less than one, the child process is not stepped. Note that the child process continues with the current signal, if any. (You can set *\$signal = 0* to prevent this.)

If you accidentally step down into a procedure you don't care about, use the *bU* command to set a temporary up-level breakpoint, and then continue using *c*.

The *S* command steps over procedure calls because *cdb* detects the occurrence of a procedure call and plants a temporary breakpoint at the point of return, free runs the program until that breakpoint is hit, then machine-instruction steps to the next source line boundary. If a breakpoint is hit during execution of the called procedure, execution stops at that point and the temporary breakpoint is deleted.

Stepping into a non-debuggable procedure (i.e., one that hasn't been compiled with *-g*) with *s* will cause behavior equivalent to *S*. In general, you can't do anything with non-debuggable code. In the stepping case, *cdb* recognizes that it has stepped into an unknown (non-debuggable) procedure, so it sets an invisible up-level breakpoint and free runs the child.

You can't specify *arguments* with *s* and *S*. If you need to specify *arguments* to redirect I/O, the easiest way is to set a breakpoint on the first line of *main()* and execute with *r*.

Example:

```
>D
All breakpoints deleted.
>s
Starting process 1089: "a.out"
main.c: main: 4: i = 5;
>s
main.c: main: 5: asub(i);
>s
sub.c: asub: 4: bsub(arg);
>S
sub.c: asub: 5: }
>2s
main.c: main: 6: }
Child process terminated normally
```

The debugger has no knowledge about or control over child processes forked in turn by the process being debugged. Also, it gets very confused (leading to **bad access messages**) if the process being debugged executes a different program via *exec(2)*.

Child process output may be (and usually is) buffered. Hence it may not appear immediately after you step through an output statement such as *printf(3)*. It may not appear at all if you kill the process.

Notes



Breakpoint Commands

The debugger provides a number of commands for setting and deleting breakpoints. A breakpoint has three attributes associated with it:

- *address* - All the commands which set a breakpoint are simply alternate ways to specify the breakpoint address. The breakpoint is then encountered whenever the instruction *address* is about to be executed, regardless of the path taken to get there. Only one breakpoint at a time (of any type or count) may be set at a given *address*. Setting a new breakpoint at *address* replaces the old one, if any.
- *count* - The number of times the breakpoint is encountered prior to recognition. If *count* is positive, the breakpoint is **permanent**, and *count* decrements with each encounter. Each time *count* goes to zero, the breakpoint is recognized, and *count* is reset to one (so it stays there until explicitly set to a different value by a *c* or *C* command).

If *count* is negative, the breakpoint is **temporary**, and *count* increments with each encounter. Once *count* goes to zero, the breakpoint is recognized, then deleted.

A *count* of zero is used internally by the debugger and means that the breakpoint is deleted when the child process next stops for any reason, whether it hit that breakpoint or not. Commands saved with such breakpoints are ignored. Normally you never see this kind of breakpoints.

Note that *count* is set to either -1 (temporary) or 1 (permanent) for any new breakpoint. It can then be modified only by the *c* or *C* command.

- *commands* - *cdb* commands which are executed when a breakpoint is recognized. These are separated by ; and may be enclosed in {} to delimit the list saved with the breakpoint from other commands on the same line. If the first character is anything other than {, or if the matching } is missing, the rest of the line is saved with the breakpoint.

Remember that the results of expressions followed by ; or } are not printed unless you specify a print format. You can use /*n* (normal format) to force printing of a result.

Saved commands are not parsed until the breakpoint is recognized. If *commands* does not exist then, after recognition of the breakpoint, the debugger waits for command input.

The debugger has only one active command line at a time. When it begins to execute breakpoint commands, the remainder (if any) of the old command line is tossed, with notice given.

Breakpoints can be set at executable statements only. By definition an **executable line** is one for which the compiler has emitted an **SLT** (Source Line Table) entry. The C compiler emits SLT entries for each logical statement (*assignment, while, for, if*, etc). If you put several assignment statement on the same source line, the compiler will emit several SLT entries for that line. You can set breakpoints only at the first SLT entry for a line, but stepping through that line with *s* will repeatedly show the same line. This is because you are hitting addresses corresponding to successive SLT entries on that line.

Attempting to set a breakpoint on a non-executable line has several possible results. If the line is before the first executable line in a procedure or after the last executable line in a file, *cdb* displays:

```
"Can't set breakpoint (invalid address)"
```

If the line is between two executable lines, *cdb* **rounds forward** and sets the breakpoint on the following executable line.

Set a Breakpoint

Syntax:

```
[line] b [commands]
```

cdb provides several commands for setting breakpoints. The simplest is *b* which sets a permanent breakpoint at the current line. The *commands* descriptor is a list of *cdb* commands, separated by semi-colons, which are executed when the breakpoint is hit. The *line* number refers to the current file. If the *line* number is omitted, the breakpoint is set on the current line.

When the breakpoint is recognized, *commands* are executed. If there are none, the debugger pauses for command input. If immediate continuation is desired, finish the command list with *c*.

For example, suppose you want to set a breakpoint in some file or procedure other than where you are at the moment. First, use the *e* command to get you to the right file or procedure. Look around for the line where you want the break to occur (using searches, or just by printing the lines). Once you are there, you can just say *b* to set a breakpoint on that line.

So to set a breakpoint in *asub()*, you must first set the current file to *sub.c*. Do this with the *e* command previously discussed:

```
e sub.c
```

or

```
e asub
```

Then set the breakpoint with the *b* command, possibly specifying a line number.

Example:

```
>e asub
sub.c: asub: 4: bsub(arg);
>b
Added:
 1: count: 1 asub: 4: bsub(arg);
>
```

or:

```
>e sub.c
sub.c: 1: asub (arg)
>4b
Added:
 1: count: 1 asub: 4: bsub(arg);
>D (to delete the breakpoint)
```

You can specify commands to be executed when a breakpoint is hit. Consider the following example in which *b t;c* plants a breakpoint in *bsub()* to print a stack trace, then continue execution:

```
>e bsub
sub.c: bsub: 11: }
>b t;c
Added:
 1: count: 1 bsub: 11: }
   {t;c}
>r
Starting process 3981: "a.out"

breakpoint at 0x60180022
sub.c: bsub: 11: }
 0 bsub (myarg = 5) [sub.c: 11]
 1 asub (arg = 5) [sub.c: 4]
 2 main () [main.c: 5]
 3 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
 4 unknown ()
Child process terminated normally
```

You can suppress the printing of the location by using the *Q* command (**quiet**) as the first in the list. If the *quiet* command appears as the first command in a breakpoint's command list, the normal announcement of *proc: line: text* is not made. This allows quiet checks of variables, etc. to be made without cluttering up the screen with unwanted output. The *Q* command is ignored if it appears anywhere else. Here's the same example as above, except it uses the *Q* command:

```
>e bsub
sub.c: bsub: 11: }
>b Q;t;c
Added:
  1: count: 1 bsub: 11: }
    {Q;t;c}
>r
Starting process 22980: "a.out"
  0 bsub (myarg = 5) [sub.c: 11]
  1 asub (arg = 5) [sub.c: 4]
  2 main () [main.c: 5]
  3 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
  4 unknown ()
Child process terminated normally
```

There are several more breakpoint setting commands with a variety of uses; they are listed below in "Miscellaneous Breakpoint Commands".

List Breakpoints

Syntax:

```
B
1 b
```

Both forms list all breakpoints in the format *num: count: nnn proc: ln: contents*, followed by *{commands}* (see the example). The leftmost number is an index number for use with the *d* (delete) command.

Example:

```
>B
  1: count: 1 bsub: 11: }
    (Q;t;c)
>1 b
  1: count: 1 bsub: 11: }
    (Q;t;c)
```

Delete Breakpoints

Syntax:

```
D[b]  
[expr] d  
D p
```

D deletes all breakpoints including **procedure** breakpoints. You can delete breakpoints one-by-one with the *d* command.

The version *d* deletes the breakpoint at the current line or the breakpoint number *expr*. If *expr* is absent, delete the breakpoint at the current line, if any. If there is none, the debugger executes a *B* command instead. Be careful; the breakpoints may be renumbered after each *d* command.

The *D p* command deletes all **procedure** breakpoints. All breakpoints set by commands other than *bp* will remain set.

Example:

```
>D  
All breakpoints deleted  
>4  
4:      bsub(arg);  
>b  
Added:  
1: count: 1 asub: 4 bsub(arg);  
>d  
Deleted:  
1: count: 1 asub: 4 bsub(arg);  
>b  
Added:  
1: count: 1 asub: 4 bsub(arg);  
>11  
11: }  
>b  
Added:  
2: count: 1 bsub: 11: }  
>2d  
Deleted:  
2: count: 1 bsub: 11: }  
>Dp  
No procedure breakpoints
```

Miscellaneous Breakpoint Commands

`bp[commands]`

Set permanent breakpoints at the beginning (first executable line) of every debuggable procedure. When any procedure breakpoint is hit, *commands* are executed.

It is permissible to set other permanent or temporary breakpoints at the same locations as these **procedure** breakpoints. If a procedure and non-procedure breakpoint are both hit at the same location, the non-procedure breakpoint has priority; the effect is the same as if there were no procedure breakpoint. It is not possible to alter the *count* of a procedure breakpoint. Procedure breakpoints must be activated and deleted as a group; it is not possible to set or delete individual ones.

Procedure breakpoints are useful for procedure stepping and tracing. For example, the command:

```
bp Q;1t;c
```

sets up procedure tracing by printing the current procedure at each breakpoint.

For the following commands, if the second character is upper case, e.g. *bU* instead of *bu*, then the breakpoint is temporary (*count* is -1), not permanent (*count* is 1).

`[depth]bb[commands]`
`[depth]bB[commands]`


Set a breakpoint at the beginning (first executable line) of the procedure at the given stack *depth*. If *depth* is not specified, use the currently viewed procedure, which might not be the same as the one at *depth* zero.

`[depth]bx[commands]`
`[depth]bX[commands]`

Set a breakpoint at the exit (last executable line) of the procedure at the given stack *depth*. If *depth* is not specified, use the currently viewed procedure, which might not be the same as the one at *depth* zero. The breakpoint is set at a point such that all returns of any kind go through it.

`[depth]bu [commands]`
`[depth]bU [commands]`

Set an up-level breakpoint. The breakpoint is set immediately after the return to the procedure at the specified stack *depth* (default one, not zero). A *depth* of zero means current location, e.g. `0bU` is a way to set a temporary breakpoint at the current value of `$pc`.




```
[depth]bt[proc][commands]
[depth]bT[proc][commands]
```

Trace current procedure (or procedure at *depth*, or *proc*). This command sets breakpoints at both the entrance and exit of a procedure. By default, the entry breakpoint *commands* are `q;2t;c`, which show the top two procedures on the stack and continues. The exit breakpoint is always set to execute `q;L;c`, which prints the procedure's return value and continues.

If *depth* is given, *proc* must be absent or it is taken as part of *commands*. If *depth* is missing but *proc* is specified, the named procedure is traced. If both *depth* and *proc* are omitted, the current procedure is traced, which might not be the same as the one at *depth* zero.

If *commands* are present, they are used for the entrance breakpoint, instead of the default shown above.



```
address ba[commands]
address bA[commands]
```

Set a breakpoint at the given code address. Note that *address* can be the name of a procedure or an expression containing such a name. Of course, if the child process is stopped in a non-debuggable procedure, or in prologue code (before the first executable line of a procedure), things may seem a little strange.

The next two commands, while not strictly part of the breakpoint group, are used almost exclusively as arguments to breakpoints (or assertions).

```
if [expr]
{commands}{{commands}}
```

If *expr* evaluates to a non-zero value, the first group of commands (the first `{}` block) is executed, else it (and the following `{`, if any) is skipped. In general, all other `{}` blocks are always ignored (skipped), except when given as an argument to an *a*, *b*, or *!* command. The *if* command is nestable, and may be abbreviated to *i*.

```
"any string you like"
```

Print the given string, which may have the standard backslashed character escapes in it, including `\n` for newline. This command is useful for labelling output from breakpoint commands.

Notes



Assertion Control Commands and Signal Handling Commands **5**

Assertion Control Commands

Assertions are lists of commands that are executed **before every statement**. This means that, if there is even one active assertion, the program is single stepped at the machine-instruction level. In other words, it runs very slowly. The primary use for assertions is tracking down nasty bugs, that result from someone corrupting a global variable. Each assertion is individually activated or suspended, in addition to the overall assertions mode.

Create New Assertion

Syntax:

a [*commands*]

To create a new assertion with a given *commands* list, which is not parsed until it's executed, use the *a* command. As with breakpoints, the *commands* list may be enclosed in {} to delimit it from other commands on the same line. Use the *l a* command to list all current assertions and the overall mode.

The debugger has only one active command line at a time. When it begins to execute assertion commands, the remainder (if any) of the old command line is tossed, with notice given.

Example:

```
>a if ($in main) {L;i/n}
Overall assertions state: ACTIVE
0: Active {if ($in main) {L; i/n}}
```

This code sets an assertion that checks if the next executable statement is in *main()*. If that statement is in *main()*, then it is displayed, along with the value of *i* in normal format. If the next executable statement is not in *main()*, nothing is displayed.

Modify an Assertion

Syntax:

```
[expr] a (a | d | s)
```

Modify the assertion numbered *expr*: activate it, delete it, or suspend it. Suspended assertions continue to exist, but have no effect until reactivated.

Example:

```
>e main
main.c: main: 4: i = 5;
>a if ($in main) {L;i/n}
Overall assertions state: ACTIVE
  0: Active {if ($in main) {L; i/n}}
>Oad                                     (delete the mistyped assertion above)
Assertion 0 deleted
>l a
No assertions
>a if ($in main) {L;i/n}
Overall assertions state: ACTIVE
  0: Active {if ($in main) {L; i/n}}
>r
Starting process 27700: "a.out"
main.c: main: 4: i = 5;
i = 0
main.c: main: 5: asub(i);
i = 5
main.c: main: 6: }
i = 5
Child process terminated normally
>Oad
Assertion 0 deleted
```

The *a* command can be used to trace variable values. For example, it can be used to trace the variable *i* which is in *main* but not known in *asub()* or *bsub()*.

Example:

```
a if (abc != $abc) {$abc = abc; abc/d; if (abc > 9) {x}}
```

This command sets up an assertion to report the changing value of some global variable (*abc*), and to stop if it ever exceeds some value. It uses a debugger local variable (*\$abc*) to keep track of the value of *abc*.

Tracing Program Execution

a L

Syntax:

a L.

This just traces execution a line at a time until something happens (e.g., you hit the **Break** key). Output from running program with above assertion. The example below illustrates setting a flag indicating whether *bsub()* has been called. It echos the flag value at every statement.

Example:

```
>a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
Overall assertions state: ACTIVE
  0: Active    {L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n}
>$bsubcalled=0
$bsubcalled = 0
>r
Starting process 27718: "a.out"
main.c: main: 4: i = 5;
$bsubcalled = 0
main.c: main: 5: asub(i);
$bsubcalled = 0
sub.c: asub: 4: bsub(arg);
$bsubcalled = 0
sub.c: bsub: 11: }
$bsubcalled = 1
sub.c: asub: 5: }
$bsubcalled = 1
main.c: main: 6: }
$bsubcalled = 1
Child process terminated normally
```

Toggle the State

Syntax:

A

Toggle the overall state of the assertions mechanism between *active* and *suspended*.

Example:

```
>A
Assertions are SUSPENDED
>r
Terminating process 1299
Starting process 1300: "a.out"

breakpoint at 0x6010000a
main.c: main: 5: asub(i);
>A
Assertions are ACTIVE
>r
Terminating process 1299
Starting process 1300: "a.out"
main.c: main: 4: i = 5;
```

Delete All Assertions

Syntax:

D a

Delete all assertions.

Example:

```
>D a
All assertions deleted
```

Certain commands (*r*, *R*, *c*, *C*, *s*, *S*, and *k*) are not allowed while assertions are running. They must appear after the *x*, if at all (see “Display Formats”).

Signal Handling Commands

The debugger catches all signals bound for the child process before the child process sees them. (This is a function of the *ptrace(2)* mechanism.) For many signals, this is reasonable. Most processes are not set up to handle segmentation errors, etc. Other processes do quite a bit with signals and the constant need to continue from a signal catch can be tedious. It is possible to alter this behavior for any or all signals.

There are three signal action attributes in the debugger:

- *cdb* can have the child process ignore or not ignore a signal. This determines whether the child process sees the signal.
- *cdb* can report or not report on when a child process receives a signal. For example, *cdb* prints out the line it occurred on.
- *cdb* can stop or not stop when a child process receives a signal.

Each above attribute is independent of the other two, yet six combinations are legal.

For this section a different program is required since *main.c* does not send or receive signals. Type this new program in the file *sig.c*:

```
main()
{
    long i,j;
    i = 5;
    j = i/0;
}
```

To compile and run the program type:

```
cc -g -o sig sig.c
sig
```

Start the *cdb* debugger by entering:

```
cdb sig
```

Reverse Handling of Signal

Syntax

`[signal] z [i][r][s][Q]`

The `z` command maintains the *signal* (signal) handling table. The variable *signal* is a valid signal number (the default is the current signal). The options (which must be all one word) toggle the state of the appropriate flag: ignore, report, or stop. If *Q* is present, the new state of the signal is not printed.

The sequence `! z` is used to list the current handling of all signals. The sequence `8 z` will only report on signal 8. Note that just `z` with no options tells you the state of the current or selected signal.

To toggle the state of a signal, type `signal z` and the actions to toggle. For example, assuming a start up state of: do stop, don't ignore, and do report, the command `8 z sir` tells the debugger to not stop, do ignore and do not report on signal 8. The command `8 z ir` toggles *Ignore* to *No* and *Report* to *Yes*. Doing `8 z ir` again toggles the flags back to the previous state.

When the debugger ignores a signal, the child process does not receive that signal.

Example:

```
cdb sig
Source files: 2
Procedures: 2
sig.c: main: 4: i = 5;
>l z
Sig  Stop  Ignore  Report  Name
  1  Yes   No      Yes     hangup
  2  Yes   Yes     Yes     interrupt
  ...
  8  Yes   No      Yes     floating point exception
  ...
 19  Yes   No      Yes     power fail
>8 z                               (list current state of signal 8)
Sig  Stop  Ignore  Report  Name
  8  Yes   No      Yes     floating point exception
>r
Starting process 11827: "sig"

floating point exception (no ignore) at 0x6010000e
sig.c: main: 5 +0x00000004: j = i/0;
>8 z sir                             (reverse the handling of signal 8)
Sig  Stop  Ignore  Report  Name
  8  No    Yes     No      floating point exception
```

```

>r
Terminating process 11827
Starting process 11873: "sig"
Child process terminated normally
>8 z ir
Sig  Stop  Ignore  Report  Name
  8   No   No      Yes     floating point exception
>r
Starting process 11891: "sig"

floating point exception (no stop) (no ignore) at 0x6010000e
sig.c: main: 5 +0x00000004: j = i/0;
floating point exception (core dumped) (no ignore) at 00000000
(file unknown): unknown: (line unknown)
Child process terminated on signal
>8 z ir
Sig  Stop  Ignore  Report  Name
  8   No   Yes     No      floating point exception

```


Notes



Record, Playback and Other cdb Commands

6

Record and Playback Commands

The debugger supports a record-and-playback feature to help re-create program states and to record all debugger output. It is particularly useful for bugs requiring long setups. With playback, you can automatically re-create a program state that may take a long time to re-construct.

The `-r` (record) and `-p` (playback) options specify record and playback files that the debugger will use. The example below sets up a scenario similar to that in the “Tracing Program Execution” section, with several other command lines entered after the command `cdb a.out -r record1`.

Example:

```
cdb a.out -r record1
Source files:  3
Procedures:   4
Recording is ON, overwriting "record1"
main.c: main: 4: i=5;
>a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
Overall assertions state: ACTIVE
  0: Active    {L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n}
>$bsubcalled=0
$bsubcalled = 0
>r
Starting process 27718:  "a.out"
main.c: main: 4: i = 5;
$bsubcalled = 0
main.c: main: 5: asub(i);
$bsubcalled = 0
sub.c: asub: 4: bsub(arg);
$bsubcalled = 0
sub.c: bsub: 11: }
$bsubcalled = 1
sub.c: asub: 5: }
$bsubcalled = 1
main.c: main: 6: }
$bsubcalled = 1
Child process terminated normally
>e asub
sub.c: asub: 4: bsub(arg);
```

```

>b t;c
Added:
  2: count: 1 asub: 4: bsub(arg);
    {t;c}
>l b
  1: count: 0 (temporary) start +0x00000024: (line unknown)
  2: count: 1 asub: 4: bsub(arg);
    {t;c}
>A
Assertions are SUSPENDED
>l a
Overall assertions state: SUSPENDED
  0: Active      {L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n}
>q
Really quit? y
$

```

All these commands are now saved in the file *record1*:

```

a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
$bsubcalled=0
r
e asub
b t;c
l b
A
l a
q
y

```

Cdb can then be exited and returned to by:

```

cdb a.out -p record1

```

The debugger re-runs all the commands and thereby re-creates the original environment.

You can also save the instructions from inside *cdb* using the `> record1` command as the first statement of the session. The sequence of commands typed in immediately after is saved in *record1*. Now instead of quitting *cdb*, the record file can be closed and started as a playback using `>c` followed by `< record1`. The commands saved in *record1* are then re-executed and the results printed to the screen. The `<<` command plays back *record1* in single step mode and provides the specialized set of instructions (see the example).

Example:

```
cdb sig
Source files: 2
Procedures: 2
sig.c main: 4: i = 5;
> > record1
Recording is ON, overwriting "record1"
>b t;c
Added:
  1: count: 1 main: 4: i = 5;
    {t,c}
>r
Starting Process 13597: "a.out"

breakpoint at 0x60100006
main.c: main: 4: i = 5;
  0 main ()      [main.c: 4]
  1 start +0x0000001a (0x11, 0xc0000030, 0xc0000040)
  2 unknown ()
Child process terminated normally
> > c
Closing record file "record1"
> < record1      (start playback from file "record1")
Playing back from "record1"
b t;c
Added:
  1: count: 1 main: 4: i = 5;
    {t,c}
r
Starting Process 13597: "a.out"

breakpoint at 0x60100006
main.c: main: 4: i = 5;
  0 main ()      [main.c: 4]
  1 start +0x0000001a (0x11, 0xc0000030, 0xc0000040)
  2 unknown ()
Child process terminated normally
>c
End of playback
> << record1      (single step, with instructions, playback)
Playing back from "record1"
b t;c (<cr>, S, <num>, C, Q, or?): ?
<cr> execute one command line;
S skip one command line;
<num> execute number of command lines;
C continue through all playback;
Q quit playback mode.
1
```

Deleted:

```
1: count: 1 main: 4: i = 5;
   {t;c}
```

Added:

```
1: count: 1 main: 4: i = 5;
   {t;c}
```

```
r (<cr>, S, <num>, C, Q, or ?): Q
End of playback
```

Miscellaneous Record and Playback Commands

The rest of the record and playback commands are used in the same manner with slight variations. The syntax and a brief description of each is listed below:

>file This command sets or changes the recordfile to *file* and turns recording on. Any previous contents of *file* are overwritten. Only commands are recorded to this file.

>>file The same as **>file**, but appends to *file* instead of overwriting.

>@file

>>@file Set or change record-all file to *file*, for overwriting or appending. The record-all file may be opened or closed independently of (in parallel with) the recordfile. All debugger standard output is copied to the record-all file, including prompts, commands entered, and command output. However, child process output is not captured.

>(t | f | c) Turn recording on (t) or off (f), or close the recording file (c). When recording is resumed, it appends after commands recorded earlier. In this context, **>>** is the same as **>**.

>@(t | f | c) Turn record-all on, off, or close the record-all file. In this context, **>>@** is the same as **>@**.

> Display the current recording status. **>>** does the same thing.

>@ Display the current record-all status. **>>@** does the same thing.

Only command lines read from the keyboard or a playback file are recorded in the **recordfile**. For example, if recording is turned on in an assertion, it doesn't take affect until assertion execution stops. Both the commands and resulting output are recorded in the **record-all file**.

Command lines beginning with `>`, `<`, or `!` are not copied to the current recordfile (but they are copied to the record-all file). You can override this by beginning the lines with blanks.

Other Commands

Two options that were not covered previously are:

- `-S` – size of cache option sets the size of the string cache to the given number of bytes, instead of the default.
- `-u` – unique names option tells the debugger to expect names in the symbol table to start with an extra underscore.

Each of the following commands are fairly straightforward. Therefore, only the syntax and a brief description of each is provided:

`Return`

Repeat the previous command

`^D`

To repeat one command 10 times use `CTRL D`

`!` *[command_line]*

This shell escape invokes a shell program in the same manner as *vi(1)*.

`f` *[printf-style-format]*

Set address printing format (the default is reset), using *printf(3)* format specifications (**not** debugger format styles).

`h` | `help`

Print the debugger help file (command summary).

`I`

Print information (inquire) about the state of the debugger.

`q`

Quit the debugger. To be sure you don't lose a valuable environment, this command requests confirmation.

`z`

Toggle case sensitivity in searches. This affects everything: file names, procedure names, variables, and string searches! The debugger starts out as **not** case sensitive.

`g` *line*

Go to an address in the procedure on the stack at *depth* zero (not necessarily the same as the currently viewed procedure).

Notes



Index

a

a	43-44
A	46
a L	45
a.out	6,14,16
adb	4
Alternate Address Map	27
Assertion Control Commands:	
a	43-44
A	46
a L	45
D a	46
Section	43-46,45
asub()	24,37,44

b

b	36-38
Breakpoint Commands:	
Address	35
b	36-38
bu,bU,bp,bb,bB,bx,bX,bt,bT,ba,bA	32,40
Commands	35
Count	35
D, d, D p	39
l b, B	38
Miscellaneous	40
Breakpoint:	
Permanent	31,35
Temporary	31,35
bsub()	37,44,45
bu,bU,bp,bb,bB,bx,bX,bt,bT,ba,bA	32,40

C

C Language	4,5,10-12,14
C, c	31-32
Case Sensitivity	8
cdb(1)	3,4,5,14
cdb:	
Debugger	4,14,21,29,30,31,36,47
Definition	4
Overview	4
Prompt	15
Running	14
Signal Action Attributes	47
Commands:	
!	55
+/-	20
+/-, w/W	21
/,?	21
>, >>, >@, >file	54-55
a	43-44
A	46
a L	45
adb	5
b	36-38
bu, bU, bp, bb, bB, bx, bX, bt, bT, ba, bA	32,40
C, c	31-32
cdb	4,5,14
D a	46
D, d, D p	39
dir	21
e	17
E	23
f	55
fdb	14
g	55
h help	55
I	55
if	41
k	30
L	21,23
l	26
l b, B	38
line	21

lsf	16
n, N	21
OE	23
Other	55
p	18
pdb	14
q	55
R, r	29-30
S, s	32-33
Search	21
t, T	22
w	19-20
z	48-49
Z,	55
^D	55
“any string you like”	41
Compilers:	
C	5,47
FORTRAN	5
Pascal	5
Compiling Options	5-6
Compiling Programs	5-6,47
Continue After Breakpoint/Signal	31-32
Conventions:	
Expression	10-12
Manual	3
Notational	7-6
Procedure Call	13
Variable Name	8-9
Core	6
Corefile	6,13
count	27
creat(2)	29
Create New Assertion	43

d

D a	46
D, d, D p	39
Data Viewing Commands:	
Display Formats	27-28
expr	24
l	26
Miscellaneous	26
proc.var, proc.depth.expr	25
Section	25-26
Debugger:	
adb	5
cdb	4,5,14
fdb	4,14
pdb	4,14
Delete Assertions	46
Delete Breakpoints	39
depth	22
dir	21
Display Formats	27
Dot (.)	8,26

e

E	23
Editor vi	15
end.c	13
Example Programs	15,47
exec(2)	33
expr	24
Expression Conventions	10-12

f

f	55
fdb	14
File Code Viewing Commands:	
+/-	20
e	17
Miscellaneous	21
p	18
Section	17-21
w	19-20
Files:	
/usr/lib/end.o	6
a.out	4,14,16
Core	6
Corefile	4,13
end.c	13
Objectfile	5,6,13
Playback	6,51
Record	6,51
Record-all File	54
Recordfile	54
sig.c	47
sub.c	37
format	8,26,27
FORTRAN	4,5,10-12,14
Free run	4

g

g	55
---------	----

h

h help	55
HP-UX Reference	3,4

i

I	55
if	41
Interprocess Debugger	4

j

Job Control Commands:	
C, c	31-32
k	30
R, r	29-30
S, s	32-33
Section	29-33

k

k	30
---------	----

I

L	21,23
l b, B	38
Languages:	
C	4,5,10-12
FORTRAN	4,5,10-12,13
Pascal	4,5,10-12
line	21
List Breakpoints	38
List Command	26
lsf	16

m

main()	15,24,33,43
main.c	15-16,47
Manual Page:	
cdb(1)	4,5,14
ptrace(2)	5,47
Manual:	
Conventions	3
HP-UX Reference	3-4
Organization	1
Miscellaneous:	
Breakpoint Commands	40
Data Viewing Commands	26
File Code Viewing Commands	21
Record, Playback Commands	54-55
Modifiers:	
command	8
depth	7
expr	7
file	7
format	8,26,27
number	7
proc	7
var	7
Modify Assertion	44
Move Forward/Backward	20

n

n, N	21
Notational Conventions	7-6

o

Objectfile	6,7,13
open(2)	29
Options:	
-d	6,14
-g	5,13,16
-l	5
-p	6,51,52
-r	6,51
-s	6
-u	13

p

p	18
Pascal	4,5,10-12,14
pdb	14
Permanent Breakpoint	31,35
Playback File	6,51
Playback Single Step Mode	52-54
Playback the Commands	52-54
Print:	
Current File, Procedure, Line#	17
Groups of Lines	18
Variable's Value	24
Window of Text	19-20
printf(3)	13,33
proc.depth.var	23,25
proc.var	8,25
Procedure Call Conventions	13
Program:	
main.c	15-16,47
sub.c	15-16
Programs:	

Compiling	5-6,47
Example	15,47
Invoking Debugger on	14
main.c	15,47
sig.c	47
ptrace(2)	4,47

q

q	55
---------	----

r

Record File	6,51
Record the Commands	51-52
Record, Playback Commands:	
< record1	52-54
<<	52-54
> record1	52-54
>, >>, >@, >file	54-55
>c	52-54
Miscellaneous	54-55
Options	51
r	51-53
Section	51-55
Record-all File	54
Recordfile	54
Reverse Handling of Signal	48-49
Run/Terminate Program	29-30

S

S, s	32-33
Search Commands	21
Set Breakpoint	36-38
Set Viewing Location	23
sig.c	47
Signal Handling Commands:	
Section	47-49
z	48-49
Single Step After Breakpoint	32-33
SLT (Source Line Table)	36
Special Variables	9
Stack Viewing Commands:	
E	23
Section	22-23
t, T	22
stdin	29
stdout	29
sub.c	15-16,37
Subroutine:	
asub()	37
bsub()	37,44,45
Subroutines:	
asub()	15,24,44
bsub()	15,44,45
printf(3)	13,33
stdin	29
stdout	29
System Call:	
creat(2)	29
exec(2)	33
open(2)	29
ptrace(2)	5,47

t

T,t	22
Temporary Breakpoint	31,35
Terminate Current Child Process	30
Toggle State	46
Trace Stack for Expr Level	22
Tracing Program Execution	45

v

Variable Name Conventions	8-9
Variables:	
\$Bad	10
\$lang	10,14
\$line	10
\$malloc	10
\$pc, \$fp, \$sp, \$ro, etc.	9
\$result	9
\$signal	9,32
\$var	9
:var	8
dot(.)	8,26
proc.depth.var	8,25
proc.var	8,25
Special	9
var	8
vi	15
View Non-current Location Variables	25
Viewing Commands:	
Data	24-26
File Code	17-21
Stack	22-23

z

z	48-49
Z	55

Notes



Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.



Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

HP-UX Concepts and Tutorials
Vol. 3: Software Development Tools

97089-90040

April 1985

Last Date: _____

(See the Printing History in the front of the manual)

Name: _____

Company: _____

Address: _____

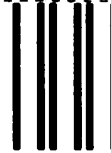
Phone No: _____

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

**HP-UX Concepts and Tutorials
Vol. 3: Software Development Tools**

97089-90040

April 1985

Last Date: _____

(See the Printing History in the front of the manual)

Name: _____

Company: _____

Address: _____

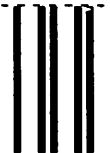
Phone No: _____

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



Manual Comment Card

If you have any comments or questions regarding this manual, write them on this comment card and place it in the mail. Include page numbers with your comments wherever possible. Enter the last date from the Printing History page on the line above your name. Also include a return address so that we can respond as soon as possible.

HP-UX Concepts and Tutorials
Vol. 3: Software Development Tools

97089-90040

April 1985

Last Date: _____

(See the Printing History in the front of the manual)

Name: _____

Company: _____

Address: _____

Phone No: _____

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

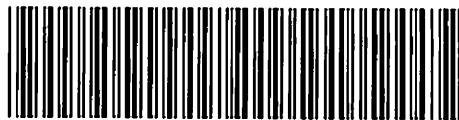
Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES







Reorder Number
97089-90040

Printed in U.S.A. 4/85

97089-90602

Mfg. No. Only