sun
microsystems

Debugging Tools Manual

Part Number: 800-3849-10
Revision A of 27 March, 1990

Trademarks

Sun Workstation® is a trademark of Sun Microsystems, Incorporated.

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

Contents

Tables

Figures

1

Introduction

1.1. Three Debuggers

This manual describes three debuggers available on Sun Workstations™: dbx, dbxtool, and adb. This document is intended for C, assembler, FORTRAN, Modula-2, and Pascal programmers.

dbx

dbx is an interactive, line-oriented, source-level, symbolic debugger. It lets you determine where a program crashed, view the values of variables and expressions, set breakpoints in the code, and run and trace a program. In addition, machine-level and other commands are available to help you debug code. A detailed description of how to use dbx is found in Chapter 4 .

dbxtool

dbxtool is a window-based interface to dbx. Debugging is easier because you can use the mouse to enter most commands from redefinable buttons on the screen. You can use any of the standard dbx commands in the command window. A detailed description of how to use dbxtool is found in Chapter 3 .

adb

adb is an interactive, line-oriented, assembly-level debugger. It can be used to examine core files to determine why they crashed, and provides a controlled environment for program execution. Since it dates back to UNIX† Version 7, it is likely to be available on UNIX systems everywhere. Chapters 6 and 7 are tutorial introductions to adb for the Sun-3 and the Sun386*i*, respectively, and Chapter 8 is a reference manual for it.

This manual begins with material about the debuggers of choice, dbxtool and dbx. They are much easier to use than adb, and are sufficient for almost all debugging tasks. adb is most useful for interactive examination of binary files without symbols, patching binary files or object code, debugging programs when the source code is not at hand, and debugging the kernel.

Some programs produce core dumps when an internal bug causes a system fault. You can usually produce a core dump by typing CTRL-\ while a process is running. If a process is running in the background, or originated from a different process group, you can get it to dump core by using the gcore(1) utility.

† UNIX is a registered trademark of AT&T.

2

dbx and dbxtool Compared

2.1. Debugging Modes of dbx and dbxtool

Both dbx and dbxtool support three distinct types of debugging: post-mortem, live-process, and multiple-process. References to dbx below apply to dbxtool as well.

You can do post-mortem debugging on a program that has created a core file. Using the core file as its image of the program, dbx retrieves the values of variables from it. The most useful operations in post-mortem debugging are getting a stack trace with where, and examining the values of variables with print. Operations such as setting breakpoints, suspending and continuing execution, and calling procedures, are not supported with post-mortem debugging.

In live-process debugging, a process's execution is controlled by dbx. From there, the user can:

- set the process' starting point
- set and clear breakpoints
- restart a stopped process.

The most useful operations are getting a stack trace with where, examining the values of variables with print and display, setting breakpoints with stop, and continuing execution with next, step, and cont.

Multiple-process debugging is most useful when debugging the interaction between two tightly coupled programs. For example, in a networking situation it is common to have server and client processes that use some style of inter-process communication (remote procedure calls, for example). To debug both the client and the server simultaneously, each process must have its own instance of dbx. When using dbx for multiple-process debugging, it is advisable to begin each dbx in a separate window. This gives you a way to debug one process without losing the context of the other debugging session.

NOTE *This does not mean that either* dbx *or* dbxtool *supports remote debugging. You can debug only processes running on your machine.*

2.2. Common Features of dbx and dbxtool

The following symbols and conventions apply to both dbx and dbxtool; as before, references to dbx apply to dbxtool as well.

Filenames

Filenames within `dbx` may include shell metacharacters. The shell used for pattern matching is determined by the `SHELL` environment variable.

Expressions

Expressions in `dbx` are combinations of variables, constants, procedure calls, and operators. Hexadecimal constants begin with "`0x`" and octal constants with "`0`". Character constants must be enclosed in single quotes. Expressions cannot involve literal strings, structures, or arrays, although elements of structures and arrays may be used. However, the `print` and `display` commands do accept structures or arrays as arguments and, in these cases, print the entire contents of the structure or array. The `call` command accepts literal strings as arguments, and passes them according to the calling conventions of the language of the routine being called.

Table 2-1 *Operators Recognized by* `dbx`

Operators Recognized by `dbx`	
`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`div`	integer divide
`%`	remainder
`<<`	left shift
`>>`	right shift
`&`	bitwise and
`|`	bitwise or
`^`	exclusive or
`~`	bitwise complement
`&`	address of
`*`	contents of
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to
`!`	not
`&&`	logical and
`||`	logical or
`sizeof`	size of a variable or type
(*type*)	type cast
`.`	structure field reference
`->`	pointer to structure field reference

The operator "." can be used with pointers to records, as well as with records themselves, making the C operator "->" unnecessary (though it is supported).

Precedence and associativity of operators are the same as in C, and are described in Table 2-2 below. Parentheses can be used for grouping.

Table 2-2 *Operator Precedence and Associativity*

<table>
<tr><th>Operator</th><th>Associativity</th><th>Precedence</th></tr>
<tr><td>. -></td><td>left to right</td><td>highest</td></tr>
<tr><td>~ ! (type) * & sizeof</td><td>right to left</td><td></td></tr>
<tr><td>* / % div</td><td>left to right</td><td></td></tr>
<tr><td>+ -</td><td>left to right</td><td></td></tr>
<tr><td><< >></td><td>left to right</td><td></td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td></td></tr>
<tr><td>== !=</td><td>left to right</td><td></td></tr>
<tr><td>&</td><td>left to right</td><td></td></tr>
<tr><td>^</td><td>left to right</td><td></td></tr>
<tr><td>|</td><td>left to right</td><td></td></tr>
<tr><td>&&</td><td>left to right</td><td></td></tr>
<tr><td>||</td><td>left to right</td><td></td></tr>
<tr><td>?:</td><td>right to left</td><td>lowest</td></tr>
</table>

Of course, if the program being debugged is not active and there is no `core` file, you may only use expressions containing constants. Procedure calls also require that the program be active.

`dbx` Scope Rules

`dbx` uses two variables to resolve scope conflicts: `file` and `func` (see Section 4.8). The values of `file` and `func` change automatically as files and routines are entered and exited during execution of the user program. They can also be changed by the user. Changing `func` also changes the value of `file`; however, changing `file` does not change `func`.

The `func` variable is used for name resolution, as in the command `print grab` where `grab` may be defined in two different routines. The search order is:

1) Search for `grab` in the routine named by `func`.

2) If `grab` is not found in the routine named by `func`, search the file containing the routine named by `func`.

3) Finally, search the outer levels — the whole program in the case of C and FORTRAN, and the outer lexical levels (in order outward) in the case of Pascal — for `grab`.

Clearly, if `grab` is local to a different routine than the one named by `func`, or is a static variable in a different file than is the routine named by `func`, it won't be found. Note, however, that `print a`grab` is allowed, as long as routine `a` has been entered but not yet exited. Note that the file containing the routine `a` might have to be specified when the file name (minus its suffix) is the same as a routine name. For example, if routine `a` is found in module `a.c`, then `print a`grab` would not be enough — you would have to use `print a`a`grab`. If in doubt as to how to specify a name, use the `whereis` command, as in `whereis grab` to display the full qualifications of all instances of the specified name — in this case `grab`.

The variable `file` is used to:

1) Resolve conflicts when setting `func` — for example, when a C program has two static routines with the same name.
2) Determine which file to use for commands that take only a source line number — for example, `stop at 55`.
3) Determine which file to use for commands such as `edit`, which has optional arguments or no arguments at all.

When `dbx` begins execution, the initial values of `file` and `func` are determined by the presence or absence of a core file or process ID. If there is a core file or process ID, `file` and `func` are set to the point of interruption. If there is no core file or process ID, `func` is set to `main` (or `MAIN` for FORTRAN) and `file` is set to the file containing `main` or (`MAIN`).

Note that changing `func` doesn't affect the place where `dbx` continues execution when the program is restarted.

3

dbxtool

dbxtool [**-kdb**] [**-I** *dir*] [*objectfile* [*corefile* | *processID*]]

dbxtool is a source-level debugger with a window and mouse-based user interface, accepting dbx's commands with a more convenient user interface. Using the mouse, one can set breakpoints, examine variable values, control execution, browse source files, and so on. There are subwindows for viewing source code, entering commands, and several other uses. This debugger functions in the suntools(1) environment, so that the standard tool manager actions, such as moving, resizing, moving to the front or back, and so on can be applied to it. For more information on dbxtool, see the dbxtool(1) man page.

In the usage above, *objectfile* is an object file produced by cc, f77, pc, or Modula-2 or a combination thereof, with the -g flag specified to produce the appropriate symbol information. If no *objectfile* is specified, one may use the debugger's debug command to specify the program to be debugged. The object file contains a symbol table which includes the names of all the source files translated by the compiler to create it. These files are available for perusal while using the debugger, and can be seen with the modules command.

NOTE *Every stage of the compilation process, including the linking and loading phases, must include the* -g *option.*

dbxtool can be used to examine the state of the program when it faulted if a file named core exists in the current directory, or a *corefile* is specified on the command line or in the debug command.

Giving a *processID* instead of a *corefile*, halts that process and begins debugging it. Detaching the debugger from the process allows that process to continue to execute.

3.1. `dbxtool` Options

–kdb

Debugs a program that sets the keyboard into up-down translation mode. This flag is necessary if you are debugging a program that uses up-down decoding.

–I *dir*

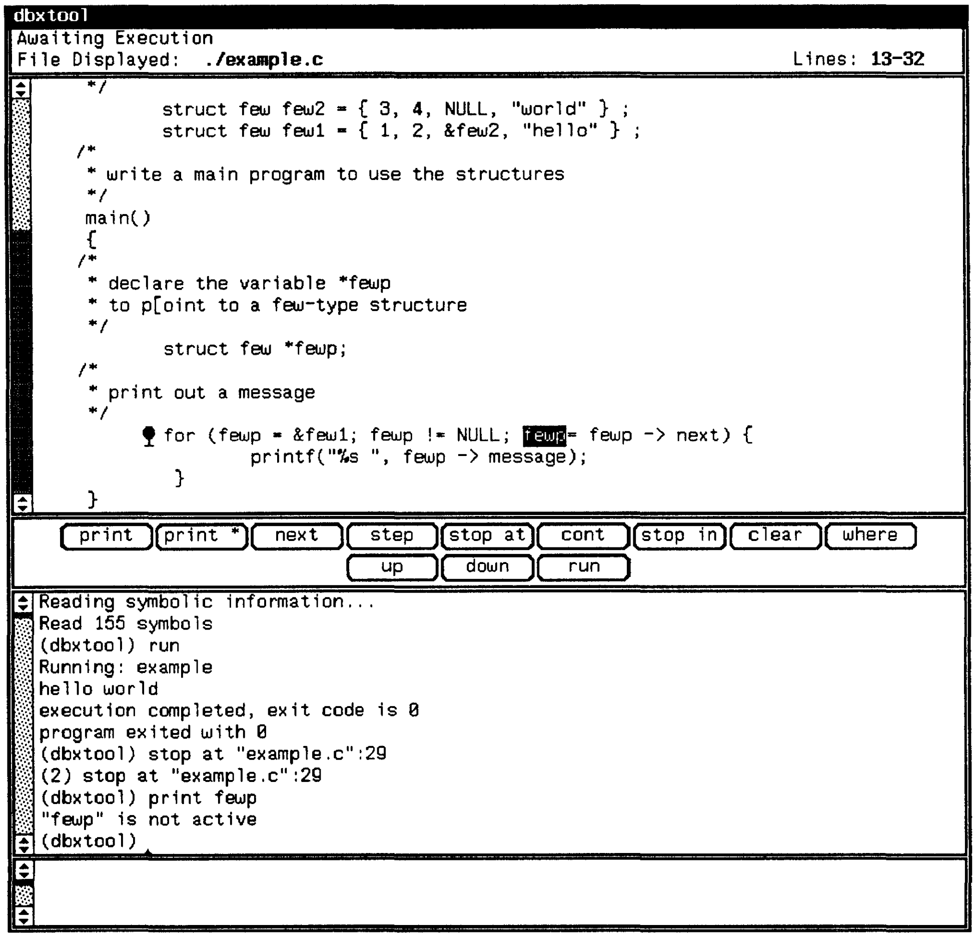
Add *dir* to the list of directories searched when looking for a source file. Normally `dbxtool` looks for source files in the directory where *objectfile* is located, and if the source files can't be found there or in the current directory, the user must tell `dbxtool` where to find the source files; either by means of the `-I` option or else by setting the directory search path by means of the `use` command. Multiple –I options may be given.

3.2. `dbxtool` Subwindows

A `dbxtool` window consists of five subwindows. From top to bottom they are:

status	Gives the overall status of debugging, including the location where execution is currently stopped, and a description of lines displayed in the *source* subwindow.
source	Displays source text of the program being debugged, and allows you to move around in the source file.
buttons	Contains buttons for frequently used commands; picking a button with the mouse invokes the corresponding command.
command	Provides a typing interface to supplement the *buttons* subwindow. Also, most command output appears in this subwindow.
display	Display output appears here.

Figure 3-1 *Five* dbxtool *Subwindows*



3.3. Scrolling

The *source*, *command*, and *display* windows have scroll bars to facilitate browsing their contents. The scroll bar is at the left edge of each window.

See the *SunView User's Guide* for a more complete description of scroll bars.

3.4. The Source Window

The *source* window displays the text of the program being debugged. Initially, it displays text from either the main routine, if there is no core file, or the point at which execution stopped, if there is a core file. Whenever execution stops during a debugging session, it displays the point at which it stopped. The `file` command can be used to switch the *source* window to another file; the focus of attention moves to the beginning of the new file. Similarly, the `func` command can be used to switch the *source* window to another function; the new focus of attention is the first executable line in the function.

Breakpoints are indicated in the *source* window by a solid stop sign at the beginning of the line. The point at which execution is currently stopped is marked by a rightward pointing outlined or hollow arrow.

3.5. Constructing Commands

One can either type commands to `dbxtool`, in the *command* window or construct commands with the selection and button mechanism (if a button is provided for the command), but typing and buttons cannot be combined to build a command.

The *command* window is a text subwindow and so uses the text selection facility described in the *SunView User's Guide*.

The software buttons operate in a postfix manner. That is, one first selects the argument, and then clicks the software button with the left mouse button. Each command interprets the selection as appropriate for that command.

There are five ways that `dbxtool` may interpret a selection:

literal — A selection may be interpreted as exactly representing selected material.

expand — A selection may be interpreted as exactly representing selected material, except that it is expanded if either the first or last character of the selection is an alphanumeric character or underscore. It is expanded to the longest enclosing sequence of alphanumeric characters or underscores. Selections made outside of `dbxtool` cannot be expanded and are interpreted as exactly the selected text.

lineno — A selection in the *source* window may be interpreted as representing the (line number of the) first source line containing all or some of the selection.

command — A selection in the *command* window may be interpreted as representing the command containing the selection.

ignore — Buttons may ignore a selection.

3.6. Command Buttons

The standard set of command buttons in the *buttons* window is as follows:

`print` — Print the value of a variable or expression. Since this button expands the selection, identifiers can be printed by selecting only one character.

`print *` — Print the value at the address represented by the selected variable or expression.

`next` — Execute one source statement and then stop execution, except that if the statement contains a procedure or function call, execute through the called routine before stopping. The `next` button ignores the selection.

`step` — Execute one source line and then stop execution again. If the current source line contains a procedure or function call, stop at the first executable line within the procedure or function. The `step` button ignores the selection.

`stop at` — Set a breakpoint at a given source line. Interpret a selection in the *source* window as representing the line number associated with the first line of the selection.

`cont` Resume execution from the point where it is currently stopped. The `cont` button ignores the selection.

`stop in` Set a breakpoint at the first line of a given function or procedure. Since this button expands the selection, identifiers may be printed by selecting only one character.

`clear` Clear all breakpoints at the currently selected point. `<lineno> clear` clears all breakpoints at the specified line number.

`where` Prints a procedure traceback. `<number> where` prints `number` top procedures in the traceback.

`up` Moves up the call stack one level. `<number> up` moves the call stack up `number` levels.

`down` Moves the call stack down one level. `<number> down` moves the call stack down `number` levels.

`run` Begins execution of the program. `<arguments> run` begins execution of the program with new arguments.

The Button Command

The `button` command defines buttons in the *buttons* window. It can be used in `.dbxinit` to define buttons not otherwise displayed, or during a debugging session to add new buttons. The first argument to `button` is the selection interpretation for the button, and the remainder is the command associated with it. The default set of buttons can be replicated by the following sequence:

```
button expand print
button expand print *
button ignore next
button ignore step
button lineno stop at
button ignore cont
button expand stop in
button ignore clear
button ignore where
button ignore up
button ignore down
button ignore run
```

The `unbutton` command may be used in `.dbxinit` to remove a default button from the *buttons* window, or during a debugging session to remove an existing button. The argument to `unbutton` is the name of the command associated with the button.

3.7. The Display Window

The *display* window provides continual feedback of the values of selected variables. The `display` command specifies variables to appear in the *display* window, and `undisplay` removes them. Each time execution of the program being debugged stops, the values of the displayed variables are updated.

3.8. Editing in the Source Window

The *source* window is a standard text subwindow (see *SunView User's Guide* for details). Initially `dbxtool` puts the source subwindow in browse mode, meaning that editing capabilities are suppressed. `dbxtool` adds a "start editing" entry to the standard text subwindow menu in the *source* window. When this menu item is selected, the file in the *source* window becomes editable, the menu item changes to "stop editing", and any annotations (stop signs and arrows) are removed. The "stop editing" menu item is a pull-right menu with two options: "save changes" and "ignore changes". Selecting either of these menu items disables editing, changes the menu item back to "start editing", and causes the annotations to return.

After editing a source file, it is advisable to rebuild the program, as the source file no longer reflects the executable program.

3.9. Controlling the Environment

The `toolenv` command provides control over several facets of `dbxtool`'s window environment, including the font, the vertical size of the *source*, *command*, and *display* windows, the horizontal size of the tool, and the minimum number of lines between the top or bottom of the *source* window and the arrow. These are chiefly useful in the `.dbxinit` file to control initialization of the tool, but may be issued at any time.

3.10. Other Aspects of `dbxtool`

The commands, expression syntax, scope rules, etc. of `dbxtool` are identical to those of `dbx`. Three of the commands, `toolenv`, `button`, and `unbutton` affect only `dbxtool`, so they are described below. See Chapter 4 for descriptions of the others.

`toolenv`

`toolenv` [*attribute* *value*]

Set or print attributes of the dbxtool window. This command has no effect in dbx. The possible *attribute-value* pairs and their interpretations are as follows:

Table 3-1 *Attribute-Value Pairs for* dbxtool

Attribute-Value	*Description*
font *fontfile*	change the font to that found in *fontfile*; default is taken from the DEFAULT_FONT shell variable.
width *nchars*	change the width of the tool window to *nchars* characters; default is 80 characters.
srclines *nlines*	make the source subwindow *nlines* high; default is 20 lines.
cmdlines *nlines*	make the command subwindow *nlines* high; default is 12 lines.
displines *nlines*	make the display subwindow *nlines* high; default is 3 lines.
topmargin *nlines*	keep the line with the arrow at least *nlines* from the top of the source subwindow; default is 3 lines.
botmargin *nlines*	keep the line with the arrow on it at least *nlines* from the bottom of the source subwindow; default is 3 lines.

The toolenv command with no arguments prints the current values of all the attributes.

button

button *selection command-name*

Associate a button in the *buttons* window with a command in dbxtool. This command has no effect in dbx. The argument *selection* may be any of literal, expand, lineno, command and ignore, as described in Section 3.5 . The *command_name* argument may be any sequence of words corresponding to a dbxtool command.

unbutton

unbutton *command-name*

Remove a button from the *buttons* window. The first button with a matching *command-name* is removed.

menu

The menu command defines the menu list in the *buttons* window. It can be used in .dbxinit to define menu items not otherwise displayed, or during a debugging session to add new menu items. The first argument to menu is the selection interpretation for the menu, and the remainder is the command associated with it. The default set of menu items can be replicated by the following sequence:

```
menu expand display
menu expand undisplay
menu expand file
menu expand func
menu ignore status
menu lineno cont at
menu ignore make
menu ignore kill
menu expand list
menu ignore help
```

unmenu

The `unmenu` command may be used in `.dbxinit` to remove a default menu item from the *menu* associated with the buttons window or, during a debugging session, to remove an existing menu item. The argument to `unmenu` is the menu item to be removed.

3.11. Bugs

The interaction between scrolling in the *source* subwindow and `dbx`'s regular expression search commands is wrong. Scrolling should affect where the next search begins, but it does not.

4

dbx

dbx [**-r**] [**-kbd**] [**-I** *dir*] [*objectfile* [*corefile* | *processID*]]

dbx is a tool for source-level debugging and execution of programs, that accepts the same commands as dbxtool, but has a line-oriented user interface, which does not use the window system. It is useful when you can't run SunView. (See also the dbx(1) man page.)

Table 4-1 dbx *Functions*

dbx *Functions*	
Function	*Commands*
list active procedures	`down, up, where`
name, display, and set variables	`assign, display, dump, print, set, set81, undisplay, whatis, whereis, which`
set breakpoints	`catch, clear, delete, ignore, status, stop, when`
run and trace program	`call, cont, next, rerun, run, step, trace`
access source files & directories	`cd, edit, file, func, list, pwd, use, /, ?`
machine-level commands	`nexti, stepi, stopi, tracei, address, +`
miscellaneous commands	`alias, dbxenv, debug, detach, help, kill, make, modules, quit, sh, source, setenv`

Although dbx provides a wide variety of commands, there are a few that you will execute most often. You will probably want to

- find out where an error occurred,

- display and change the values of variables,
- display the values of constants,
- set breakpoints,
- and run and trace your program.

4.1. Preparing Files for dbx

When compiling programs with `cc`, `f77`, or `pc`, you must specify the `-g` option on the command line, so that symbolic information is produced in the object file. Every step of compilation (including linking and loading) *must* include this option.

In the past, many `dbx` users have compiled, with the `-g` option, only those modules suspected of containing a bug that they wanted to fix, as this was an efficient means of debugging large programs. Those modules compiled without `-g` were accessible by just a few `dbx` commands, such as `stop in <procedure/function>`, `trace <procedure/function>`, and `print <global>`. However, `dbx` now contains the `modules` command, which is expressly designed to aid in the debugging of large programs.

The `modules` command allows you to specify those modules for which `dbx` should read source level debugging information. Therefore, it is recommended that most, if not all, modules be compiled with the `-g` option, and the `modules` command used to debug the resulting program. For more information on the `modules` command, see Section 4.12, "*Debugging Large Programs.*"

NOTE *The following list contains a few notes you may want to keep in mind while using* `dbx`:

- `dbx` won't correctly debug library modules whose names are more than 14 characters long. While `ar` emits a warning at the time the library is being created that the name of the file is being truncated, `dbx` will offer no warning that there is a problem, other than not working correctly as you attempt to debug the offending module.

- If you use `ld`'s `-r` option when compiling your program, attempts to debug the final load module with `dbx` will often fail. This is because `ld -r` modifies the symbol table and the resultant load module.

- `dbx` may not work on programs using shared libraries, especially user-defined shared libraries. Therefore, for best results, compile and link your programs statically, on the command line, with the `-Bstatic` option.

4.2. Invoking dbx

To invoke `dbx`, type:

```
demo% dbx options objfile corefile | processid
```

NOTE *All the arguments are optional.*

dbx begins execution by printing:

```
Reading symbolic information...
Read nnn symbols
(dbx)
```

To exit dbx and return to the command level, type:

```
(dbx) quit
demo%
```

For additional information and assistance, see Debugging Tips for Programmers in Chapter 5 where a sample FORTRAN program and several examples are provided. With a few changes and modifications to the examples this chapter may also be useful for C programmers.

4.3. dbx Options

The *options* to dbx are:

-r Execute *objfile* immediately. Arguments to the program being debugged follow the object filename (redirection is handled properly). If the program terminates successfully, dbx exits. Otherwise, dbx reports the reason for termination and waits for your response. When -r is specified and standard input is not a terminal, dbx reads from /dev/tty.

-kdb
Debugs a program that sets the keyboard into up-down translation mode. This flag is necessary if a program uses up-down decoding.

-I *dir*
Add *dir* to the list of directories searched when looking for a source file. Normally, dbx looks for source files in the directory where *objfile* is located, and if the source files can't be found there or in the current directory, the user must tell dbx where to find the source files; either by specifying the -I option or by setting the directory search path with the use command.

The *objfile* contains compiled object code. If it is not specified, one can use *dbx's* debug command to specify the program to be debugged. The object file contains a symbol table, which includes the names of all the source files the compiler translated with -g. These files are available for perusal while using the debugger.

If a file named core exists in the current directory, or a *corefile* is specified, dbx can be used to examine the state of the program when it faulted. If a *processID* is given instead, dbx halts that process and begins debugging it. If you later detach the debugger from it, the process continues to execute.

The `.dbxinit` File

Users of prior releases of `dbx` may have grown used to setting breakpoints in their `.dbxinit` file. The addition of the `modules` command has caused `.dbxinit` to be read BEFORE the symbol table information rather than AFTERWARDS as in previous versions. Hence, setting breakpoints in a `.dbxinit` file no longer works.

To work around this difficulty, you may define an alias in your `.dbxinit` file which will source another file of `dbx` commands; you can then set up this additional file to contain the breakpoint-setting commands. Once you have set up this second file with the breakpoint commands, all you need do is invoke the alias immediately after you invoke `dbx`.

The last line in `.dbxinit` may look something like this:

```
alias moredbx source .dbxinit2
```

The contents of `.dbxinit2` may look something like this :

```
stop in main
stop in initial
```

Once you have properly set up the `.dbxinit` file with the above alias, the first command you issue is:

```
moredbx
```

4.4. Listing Source Code

If you invoked `dbx` on an *objfile*, you can list portions of your program, and associated line numbers in the program's source files. For example, consider the program `example.c`, which you can see by typing:

```
(dbx) list 1,12
    1   #include <stdio.h>
    2
    3   main()
    4   {
    5       printf("goodbye world!\n");
    6       dumpcore();
    7   }
    8
    9   dumpcore()
   10   {
   11       abort();
   12   }
```

If the range of lines starts past the end of file, `dbx` will tell you the program has only so many lines; if the range of lines goes past the end of file, `dbx` will print as many lines as it can, without complaining. You can also list just a single procedure by typing its name instead of a range of lines; for example **`list main`** prints ten lines starting near the top of the `main()` procedure.

4.5. Listing Active and Post-Mortem Procedures

If your program fails to execute properly, you probably want to find out the procedures that were active when the program crashed. Use the `where` command, like this:

```
where [n]
```

`where` displays a list of the top *n* active procedures and functions on the stack, and associated sourcefile line numbers (if available). If *n* is not specified, all active procedures are displayed.

When debugging a post-mortem dump of the `example.c` program above, `dbx` prints the following:

```
demo% dbx example core
Reading symbolic information...
Read 41 symbols
program terminated by signal ABRT (abort)
(dbx)
(dbx) where
abort() at 0x80e5
dumpcore(), line 12 in "example.c"
main(0x1, 0xfffd84, 0xfffd8c), line 7 in "example.c"
(dbx)
```

Two other commands useful for viewing the stack are:

`up` [*n*]
: Move up the call stack (towards `main`) *n* levels. If *n* is not specified, the default is one. This command allows you to examine the local variables in functions other than the current one.

`down` [*n*]
: Move down the call stack (towards the current stopping point) *n* levels. If *n* is not specified, the default is one.

4.6. Naming and Displaying Data

You can name and display your data with the following commands:

`print` *expression* [, *expression* ...]
: Print the values of specified expressions. An expression may involve function calls if you are debugging an active process. If execution of a function encounters a breakpoint, execution halts and the `dbx` command level is re-entered. A stack trace with the `where` command shows that the call originated from the `dbx` command level.

Variables having the same name as one in the current function may be referenced as *funcname* ‘ *variable*, or *filename* ‘ *funcname* ‘ *variable*. The *filename* is required if *funcname* occurs in several files or is identical to a *filename*. For example, to access variable `i` inside routine `a`, which is declared inside module `a.c`, you would have to use `print a'a'i` to make the name `a` unambiguous. Use `whereis` to determine the fully qualified name of an identifier. For more details, see `dbx` Scope Rules in Chapter 5.

Hexadecimal numbers can be printed using the `alias` command in conjunction with machine-level commands. For more information, see Print in Hex in Chapter 5.

`display` [expression [, *expression* ...]]
Display the values of the expressions each time execution of the debugged program stops. The name qualification rules for `print` apply to `display` as well. With no arguments, the `display` command prints a list of the expressions currently being displayed, and a display number associated with each expression. In `dbxtool`, the variable names and values are shown in the display subwindow; in `dbx` they are printed automatically whenever execution stops.

`undisplay` *expression* [, *expression* ...]
Stop displaying the expressions and their values each time execution of the program being debugged stops. The name qualification rules for `print` apply to `undisplay` as well. A numeric expression is interpreted as a display number and the corresponding expression is deleted from the display.

`whatis` *identifier*
`whatis` *type*
Print the declaration of the given identifier or type. The identifier may be qualified with block names as above. The *type* argument is useful to print all the members of a structure, union, or enumerated type.

`which` *identifier*
Print the fully qualified form of the given identifier; that is, the outer blocks with which the identifier is associated.

`whereis` *identifier*
Print the fully qualified form of all symbols whose names match the given identifier. The order in which the symbols are displayed is not meaningful.

`assign` *variable* = *expression*
`set` *variable* = *expression*
Assign the value of the expression to the variable. Currently no type conversion takes place if the operands are of different types.

`set81` *fpreg* = *word1 word2 word3*
Treat the 96-bit value gotten by concatenating *word1*, *word2*, and *word3* as an IEEE floating-point value, and assign it to the named MC68881 floating-point register *fpreg*. Note that MC68881 registers can also be set with the `set` command, but that the value is treated as double-precision and converted to extended precision. *This command applies to Sun-3 systems only.*

`dump` [*func*]
Display the names and values of all the local variables and parameters in *func*. If not specified, the current function is used.

4.7. Setting Breakpoints

Breakpoints are set with the `stop` and `when` commands, which have the following forms:

`stop at` *source-line-number* [`if` *condition*]
    Stop execution at the given line number whenever the *condition* is true. If *condition* is not specified, stop every time the line is reached.

`stop in` *procedure/function* [`if` *condition*]
    Stop execution at the first line of the given procedure or function whenever the *condition* is true. If *condition* is not specified, stop every time the procedure or function is entered.

`stop` *variable* [`if` *condition*]
    Stop execution whenever the value of *variable* changes and *condition* is true. If *condition* is not specified, stop every time the value of *variable* changes. This command performs interpretive execution, and thus is significantly slower than most other *dbx* commands.

`stop if` *condition*
    Stop execution whenever *condition* becomes true. This command performs interpretive execution, and thus is significantly slower than most other *dbx* commands.

`when in` *procedure/function* { *command; ...* }
    Execute the given `dbx` *command(s)* whenever the specified procedure or function is entered.

`when at` *source-line-number* { *command; ...* }
    Execute the given `dbx` *command(s)* whenever the specified *source-line-number* is reached.

`when` *condition* { *command; ...* }
    Execute the given `dbx` *command(s)* whenever the *condition* is true before a statement is executed. This command performs interpretive execution, and thus is significantly slower than most other `dbx` commands.

NOTE *In the* `when` *commands, the braces and the semicolons between commands are required.*

The following commands can be used to view and change breakpoints:

`status` [>*filename*]
    Display the currently active `trace`, `stop`, and `when` commands. A *command-number* is listed for each command. The *filename* argument causes the output of `status` to be sent to that file.

`delete` *command-number* [[,] *command-number* ...]
`delete all`
    Remove the `trace`, `when`, and/or `stop` commands corresponding to the given *command-numbers*, or all of them. The `status` command explained above displays the numbers associated with these commands.

`clear` [*source-line-number*]
    Clear all breakpoints at the given source line number. If no *source-line-number* is given, the current stopping point is used.

Two additional commands can be used to set a breakpoint when a signal is detected by the program, rather than a condition or location.

`catch` [*number* [[,]*number* ...]]
: Start trapping the signals with the given *number*(s) before they are sent to the program being debugged. This is useful when a program handles signals such as interrupts. Initially all signals are trapped except `SIGHUP`, `SIGEMT`, `SIGFPE`, `SIGCONT`, `SIGCHLD`, `SIGALRM`, `SIGKILL`, `SIGTSTP`, and `SIGWINCH`. If no *number* is given, list the signals being caught.

`ignore` [*number* [[,] *number* ...]]
: Stop trapping the signals with the given *number*(s) before they are sent to the program being debugged. This is useful when a program handles signals such as interrupts. If no *number* is given, list the signals being ignored.

4.8. Running and Tracing Programs

You can run and trace your code using the following commands:

`run` [*args*] [> *filename* | >> *filename*]
: Start executing *objfile*, specified on the `dbx` command line (or with the most recent `debug` command), passing *args* as command-line arguments; <, >, and >> can be used to redirect input or output in the usual manner. Otherwise, all characters in *args* are passed through unchanged. If no arguments are specified, the argument list from the last `run` command (if any) is used. If *objfile* has been written since the last time the symbolic information was read in, `dbx` reads the new information before beginning execution. For more information, see Passing Arguments to a Main Program in Chapter 5

`rerun` [*args*] [> *filename* | >> *filename*]
: Identical to `run`, except in the case where no arguments are specified. In that case `run` runs the program with the same arguments as on the last invocation, whereas `rerun` runs it with no arguments at all.

`cont` [`at` *source-line-number*] [`sig` *sig-number*]
: Continue execution from where it stopped, or, if the clause `at` *source-line-number* is given, at that line number. The *sig-number* causes execution to continue as if that signal had occurred. The *source-line-number* is evaluated relative to the current file and must be within the current procedure/function. Execution cannot be continued if the process has finished (that is, has called the standard procedure `_exit`). `dbx` captures control when the process attempts to exit, thereby letting the user examine the program state.

`trace` *source-line-number* [`if` *condition*]
`trace` *procedure/function* [`if` *condition*]
`trace` [`in` *procedure/function*] [`if` *condition*]
`trace` *expression* `at` *source-line-number* [`if` *condition*]
`trace` *variable* [`in` *procedure/function*] [`if` *condition*]
: Display tracing information when the program is executed. A number is associated with the `trace` command, and can be used to turn the tracing off (see the `delete` command).

If no argument is specified, each source line is displayed before it is executed. Execution is substantially slower during this form of tracing.

The clause `in` *procedure/function* restricts tracing information to be displayed only while executing inside the given procedure or function. Note that the *procedure/function* traced must be visible in the scope in which the `trace` command is issued — see the `func` command.

The *condition* is a Boolean expression evaluated before displaying the tracing information; the information is displayed only if *condition* is true.

The first argument describes what is to be traced. The effects of different kinds of arguments are described below:

Table 4-2 *Tracing and its Effects*

source-line-number	Display the line immediately before executing it. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, for example, `"mumble.p":17`.
procedure/function	Every time the procedure or function is called, display information telling what routine called it, and what parameters were passed to it. In addition, its return is noted, and if it is a function, the return value is also displayed.
expression	The value of the expression is displayed whenever the identified source line is reached.
variable	The name and value of the variable are displayed whenever the value changes. Execution is substantially slower during this form of tracing.

Tracing is turned off whenever the function in which it was turned on is exited. For instance, if the program is stopped inside some procedure and tracing is invoked, the tracing will end when the procedure is exited. To trace the whole program, tracing must be invoked before a `run` command is issued.

When using *conditions* with `trace`, `stop`, and `when`, remember that variable names are resolved with respect to the scope current at the time the command is issued (not the scope of the expression inside the `trace`, `stop`, or `when` command). For example, if you are currently stopped in function `foo()` and you issue the command

```
stop in bar if x==5
```

the variable *x* refers to the *x* in function `foo()`, not in `bar()`. The `func` command can be used to change the scope before issuing a `trace`, `stop`, or `when` command, or the name can be qualified, for example, `bar.x==5`.

`step [` *n* `]`
Execute through the next *n* source lines and then stop. If *n* is not specified, it is taken to be one. Step into procedures and functions.

`next [` *n* `]`
Execute through the next *n* source lines and then stop, counting functions as single statements.

`call` *procedure* `(` *parameters* `)`
Execute the named *procedure* (or *function*), with the given *parameters*. If any breakpoints are encountered, execution halts and the `dbx` command level is reentered. A stack trace with the `where` command shows that the call originated from the `dbx` command level.

If the source file in which the routine is defined was compiled with the `-g` flag, the number of arguments is checked and warnings are issued. You must ensure that arguments of the appropriate type are passed.

If C routines are called that are not compiled with the `-g` flag, `dbx` does NOT check the number of parameters. The parameters are simply pushed on the stack as given in the parameter list.

Currently, FORTRAN alternate return points may not pass properly.

4.9. Accessing Source Files and Directories

Note: The FPA register names `$fpa0..$fpa31` can be used in arithmetic expressions and in `set` commands on machines with a FPA. This extension only applies on a machine with an FPA. Note that if an FPA register is used in an expression or assignment, its type is assumed to be double precision. FPA registers can be displayed in single precision using the `/f` display format. Double-precision values are displayed using the `/F`.

These commands let you access source files and directories without exiting `dbx`:

`edit [` *filename* `]`
`edit` *procedure/function*
Invoke an editor on *filename* (or on the current source file if none is specified). If a *procedure* or *function* name is specified, the editor is invoked on the file that contains it. The default editor invoked is `vi`. Set the environment variable `EDITOR` to the name of a preferred editor to override the default. For `dbxtool`, the editor comes up in a new window.

`file [` *filename* `]`
Change the current source file to *filename*, or print the name of the current source file if no *filename* is specified.

`func [` *procedure / function* `]`
Change the current function, or print the name of the current function if none is specified. Changing the current function implicitly changes the current source file displayed by `file` to the one that contains the function; it also changes the current scope used for name resolution.

`list [` *source-line-number* `[ ,` *source-line-number* `] ]`
`list` *procedure/function*
List the lines in the current source file from the first line number through the second. If no lines are specified, the next 10 lines are listed. If the name of a procedure or function is given, lines *n–5* to *n+5* are listed, where *n* is the first statement in the procedure or function. If the `list` command's argument is a procedure or function, the scope for further listing is changed to that routine — use the `file` command to change it back. In `dbxtool`, the

region of the file is shown in the source window and extends from the first line number to the end of the window.

`use` [*directory* ...]
Set the list of directories to search when looking for source files. If no *directory* is given, print the current list of directories. Supplying a list of directories replaces the current (possibly default) list. The list is searched from left to right.

`cd` [*dirname*]
Change `dbx`'s notion of the current directory to *dirname*. With no argument, use the value of the `HOME` environment variable.

`pwd`
Print `dbx`'s notion of the current directory.

/*string*[/]
Search downward in the current file for the regular expression *string*. The search begins with the line immediately after the current line and, if necessary, continues until the end of the file. The matching line becomes the current line.

?*string*[?]
Search upward in the current file for the regular expression *string*. The search begins with the line immediately before the current line and, if necessary, continues until the top of the file. The matching line becomes the current line.

When `dbx` searches for a source file, the value of `file` and the `use` directory search path are used. The value of `file` is appended to each directory in the `use` search path until a matching file is found. This file becomes the current file.

`dbx` knows the same filenames as were given to the compilers. For instance, if a file is compiled with the command

```
% cc -c -g ../mip/scan.c
```

then `dbx` knows the filename `../mip/scan.c`, but not `scan.c`.

4.10. Machine-Level Commands

These commands are used to debug code at the machine level:

`tracei` [*address*] [`if` *cond*]
`tracei` [*variable*] [`at` *address*] [`if` *cond*]
Turn on tracing of individual machine instructions.

`stopi` [*variable*] [`if` *cond*]
`stopi` [`at` *address*] [`if` *cond*]
Set a breakpoint at the address of a machine instruction.

`stepi`
`nexti`
Single step as in `step` or `next`, but do a single machine instruction rather than a line of source.

address, *address* / [*mode*]
address / [*count*] [*mode*]

+/ [*count*] [*mode*]
Display the contents of memory starting at the first *address* and continuing up to the second *address*, or until *count* items have been displayed. If a + is specified, the address following the one displayed most recently is used. The *mode* specifies how memory is displayed; if omitted, the last specified mode is used. The initial mode is `X`. The following modes are supported:

Mode	*Does*
`i`	display as a machine instruction
`d`	display as a halfword in decimal
`D`	display as a word in decimal
`o`	display as a halfword in octal
`O`	display as a word in octal
`x`	display as a halfword in hexadecimal
`X`	display as a word in hexadecimal
`b`	display as a byte in octal
`c`	display a byte as a character
`s`	display as a string of characters terminated by a null byte
`f`	display as a single-precision real number
`F`	display as a double-precision real number
`E`	display as an extended-precision real number

Symbolic addresses used in this context are specified by preceding a name with an ampersand `&`. Registers are denoted by preceding a name with a dollar sign `$`. Here is a list of MC680x0 register names:

Register	*Name*
`$d0-$d7`	data registers
`$a0-$a7`	address registers
`$fp`	frame pointer (same as `$a6`)
`$sp`	stack pointer (same as `$a7`)
`$pc`	program counter
`$ps`	program status

The following registers apply only to Sun-3 workstations:

Register	*Name*
`$fp0-$fp7`	MC68881 data registers
`$fpc`	MC68881 control register
`$fps`	MC68881 status register
`$fpi`	MC68881 instruction address register
`$fpf`	MC68881 flags (unused, idle, busy)
`$fpg`	MC68881 floating-point signal type

For example, to print the contents of the data and address registers in hex on a Sun-3, type `&$d0/16X` or `&$d0,&$a7/X`. To print the contents of register `d0`, type `print $d0` (one cannot specify a range with `print`). Addresses may be expressions made up of other addresses and the operators + (plus), – (minus), * (multiply), and indirection (unary *). The address may be a + alone, which causes the next location to be displayed.

See the *SPARC Architecture Reference Manual* and the *Sun-4 Assembly Language Reference Manual* for information about Sun-4 registers and addressing.

Here is the list of Sun386*i* registers:

Register	***Name***
`$ss`	stack segment register
`$eflags`	flags
`$cs`	code segment register
`$eip`	instruction pointer
`$eax`	general register
`$ebx`	general register
`$ecx`	general register
`$edx`	general register
`$esp`	stack pointer
`$ebp`	frame pointer
`$esi`	source index register
`$edi`	destination index register
`$ds`	data segment register
`$es`	alternate data segment register
`$fs`	alternate data segment register
`$gs`	alternate data segment register

On the Sun386*i*, to print the contents of the data and address registers in hex, type *&$eax/10X* or *&$eax,&$eip/X*. Data segment registers are always printed together, so *&$cs/X* is the same as *&$cs,&$gs/X*. The print command can also be as in *print $eax.*

You can also access parts of the Sun386*i* registers. Specifically, the lower halves (16 bits) of these registers have separate names, as follows:

Register	***Name***
`$ax`	general register
`$cx`	general register
`$dx`	general register
`$bx`	general register
`$sp`	stack pointer
`$bp`	frame pointer
`$si`	source index register
`$di`	destination index register
`$ip`	instruction pointer, lower 16 bits
`$flags`	flags, lower 16 bits

Furthermore, the first four of these 16 bit registers can be split into two 8-bit parts, as follows:

Register	*Name*
`$al`	lower (right) half of register $ax
`$ah`	higher (left) half of register $ax
`$cl`	lower (right) half of register $cx
`$ch`	higher (left) half of register $cx
`$dl`	lower (right) half of register $dx
`$dh`	higher (left) half of register $dx
`$bl`	lower (right) half of register $bx
`$bh`	higher (left) half of register $bx

The registers for the Sun386*i* math coprocessor are the following:

Register	*Name*
`$fctrl`	control register
`$fstat`	status register
`$ftag`	tag register
`$fip`	instruction pointer offset
`$fcs`	code segment selector
`$fopoff`	operand pointer offset
`$fopsel`	operand pointer selector
`$st0 - $st7`	data registers

4.11. Miscellaneous Commands

`sh` [*command-line*]
Pass the SunOS command line to the shell for execution. The `SHELL` environment variable determines which shell is used.

`alias` *new-command-name character-sequence*
Respond to *new-command-name* as though it were *character-sequence*. Special characters occurring in *character-sequence* must be enclosed in double quotation marks. Alias substitution as in the C shell also occurs. For example, `!:1` refers to the first argument. The command

```
alias mem "print (!:1)->mem1->mem2"
```

creates a `mem` command that takes an argument, evaluates its `mem1->mem2` field, and prints the result.

`help` [*command*]
`help`
Print a short message explaining *command*. If no argument is given, display a synopsis of all `dbx` commands.

`source` *filename*
Read `dbx` commands from the given *filename*. This is especially useful

when that file was created by redirecting a `status` command from an earlier debugging session.

`quit`
Exit `dbx`.

`dbxenv`
Set `dbx` attributes. The `dbxenv` command with no argument prints the attributes and their current values.

`dbxenv case sensitive|insensitive`
The keyword `case` controls whether upper and lower case letters are considered different. The default is `sensitive`; `insensitive` is most useful for debugging FORTRAN programs.

`dbxenv fpaasm on|off`
Controls the disassembly of FPA instructions. If you specify `off` with the `dbxenv fpaasm` command, FPA instructions are disassembled as move instructions. If you specify `on`, FPA instructions are disassembled by means of FPA assembler mnemonics. On a machine with an FPA, `fpaasm` is `on` by default. On machines without FPA, `fpaasm` is `off` by default.

Note : *All* FPA instructions are disassembled by the `off` option, not just those used in conjunction with the `fpaasm` subcommand.

`dbxenv fpabase a[ 0-7 ]| off`
Designates an MC68020 address register for FPA instructions that use base-plus-short-displacement addressing to address the FPA.

If the value is `on`, long move instructions that use the designated address register in base-plus-short-displacement mode are assumed to address the FPA, and are disassembled using FPA assembler mnemonics.

If the value is `off`, all based-mode FPA instructions are disassembled and single-stepped as move instructions. The default value of `fpabase` is `off`.

`dbxenv makeargs` *args*
The keyword `makeargs` defines which arguments will be passed to `make` when it is invoked from `dbx`.

`dbxenv speed` *seconds*
The keyword `speed` determines the interval between execution of source statements during tracing (default 0.5 seconds).

`dbxenv stringlen` *num*
The keyword `stringlen` controls the maximum number of characters printed for a `char *` variable in a C program (default 512).

`debug` [*objfile* [*corefile* / *process-id*]]
Terminate debugging of the current program (if any), and begin debugging the one found in *objfile* with the given *corefile* or live process, without incurring the overhead of reinitializing `dbx`. If no arguments are specified, the name of the program currently being debugged and its arguments are printed. You must have both the *objfile* and *corefile* or live process available to perform debugging.

`kill`
Terminate debugging of the current process and kill the process, but leave `dbx` ready to debug another. This can eliminate remains of a window program you were debugging without exiting the debugger, or allow the object file to be removed and remade without incurring a "text file busy" error message.

`modules`
Used to debug large programs. For more information see "*Debugging Large Programs*," below.

`detach`
Detach a process from `dbx` and let it continue to execute. The process is no longer under the control of `dbx`.

`setenv` *name string*
Set the environment variable *name* to the value of *string*. (See `csh`(1)).

4.12. Debugging Large Programs

The `modules` command within `dbx` helps you debug very large programs by selecting what parts of the available debugging information you want to use the next time `dbx` reads in the object file.

NOTE *To debug programs with the* `modules` *command, you must include* `main()` .

The `modules` command controls and displays the amount of source level debugging information available to `dbx`.

Usage:

`modules`
`modules SELECT` [ALL | *objname*] [*objname*]..
`modules APPEND` [*objname*] [*objname*]..

`modules` with no arguments displays the set of object files for which source level debugging information is currently available to the debugger, including the pathnames of any associated source files. If the debugger cannot access a source file for which it has debugging information, its name is followed by '`(?)`'.

Example:

```
(dbx) debug a.out
(dbx) modules
    object file(a.out)        source files
     a.o                    ../src/a.c ../src/a.y
     b.o                    ../src/b.c
(dbx)
```

Source file pathnames reflect the current search path as set by `USE` commands or the `-I` option.

The `modules` command followed by the keyword `SELECT` sets or displays the modules selection list. This list is used to control whether the debugger reads source level debugging information for a particular object file. You can use this to control the size of the dbx internal symbol tables when debugging large programs.

If the modules selection list is set and a particular object file of the executable file is not included in the list, the debugger will ignore debugging information for that file. The effect is the same as if the file had not been compiled with the -g flag.

Set the modules selection list to include specified object files with this command.

```
modules SELECT objname [ objname ] ...
```

Display the current list with the command.

```
modules SELECT
```

Before reading debugging information for a particular object file, the debugger checks whether the modules selection list is set. If it is set, the debugger compares the name of the object file against the modules selection list. If the name appears, its debugging information is read, otherwise it is ignored.

Disable the selection list with this command.

```
modules SELECT ALL
```

Once you set a modules selection list, all subsequent DEBUG commands will interrogate it. Change the list with additional

```
modules SELECT objname [ objname ] ...
```

commands.

Example:

```
demo% dbx
(dbx) debug a.out
Reading symbolic information...
Read 1600 symbols
(dbx) modules
  object file(a.out)      source files
   a.o                   ../src/a.c ../src/a.y
   b.o                   ../src/b.c
(dbx) modules select b.o
(dbx) debug a.out
Reading symbolic information...
Read 1600 symbols (1 of 2 files selected)
(dbx) modules
  object file(a.out)      source files
   b.o                   ../src/b.c
(dbx)
```

Add the named files to the modules selection list with this command.

```
modules APPEND objname [ objname ] ...
```

If the modules selection list includes any object files which do not appear in the executable being debugged, `dbx` prints a warning.

The set of object files read from an executable file may be larger than the set specified in the modules select list. To compress debugging symbols, the loader eliminates any debugging information which is redundantly defined in multiple include files (see symbol type `N_EXCL` in `<stab.h>`). If some symbols of an object file were excluded, the object file(s) where those symbols were first defined must also be read. Object files which were not selected but which were *implied* in this way are flagged by '`(*)`' in the output from the `modules` command.

Running Out of Swap Space with Large Files

If you debug large programs and do not use the `modules` command, you may run out of swap space. If so, address the problem by doing one of the following things:

- Increase the limit for the stacksize by inserting the line "`limit stacksize 8 megabytes`" into your `.cshrc` file. If 8 isn't enough, you may need 16, or even 32. But don't over do it. Start with 8.
- Make a bigger swap file. For help see the `mkfile`(8) and `swapon`(8) man pages.

  Example: Login as superuser, use the command `pstat-s` to verify your swap space usage, make the file, and tell the system to use it, as shown in the example on the following page.

```
demo#  pstat -s
6584k allocated + 512k reserved = 7096k used, 22136k
available
demo#  mkfile -nv 20m /home/swapfile
/home/swapfile 20971520 bytes
demo#  /usr/etc/swapon /home/swapfile
```

4.13. Debugging Child Processes

You may find that debugging programs with `dbx` or `dbxtool` is difficult when the program does a `fork()` and thereby creates child processes. Debugging can be done, but it does not fit into `dbx` nicely. You will have to change the source code during debugging.

Use the steps below and either `dbx` or `dbxtool` to debug programs that create child processes.

1. Insert a `sleep(20)` or a similar call in the child process path of the code which was started by the `fork()`. This delays the child code execution. There are many alternatives that can be used. You could also use `getchar()` or an infinite loop that can be broken by the `dbx` command `set`.
2. On SunOS releases prior to 4.0, link with the `-N` flag. This ensures that after the `fork()`, the child and parent processes have their own copies of the text segment for the process, rather than sharing the segment. Beginning with SunOS 4.0, this flag is not necessary due to the copy-on-write capability provided by the virtual memory subsystem.
3. Start `dbx` on the parent process. Put a break point in the parent process code as needed. Be sure to put a break in the execution path of the parent process right after the `fork()` point, in order to obtain the child process PID.

   Do *not* put any breakpoints in the child process at this point.
4. Start another copy of `dbx`, or `dbxtool`, and enter the first part of a command as shown below.

Do *not* press [Return] yet.

```
demo % dbx executable_filename
```

5. Start parent process code execution in the first `dbx`. Obtain the child process PID number after reaching the breakpoint set in step 3 above. We will use "1234" as the PID in this example.
6. Now complete the command as shown below.

```
demo% dbx executable_filename  1234
Reading symbolic information...
    ...
```

This command starts a second `dbx` process to debug the child process suspended earlier by the `sleep(20)` or functionally-equivalent command or loop. A `step` command now allows you to debug the child process 20 seconds after the `fork()` call.

7. You may want to trace one of the `exec()` calls executed by most child processes. The PID remains the same, but the executable image changes. A `sleep(20)` command in the process which was started by `exec()` will slow it down so that a `dbx` can attach to it. Use the following commands from the `dbx` of the child process in this case. Note that the child process will now execute at full speed.

```
demo% detach
demo% dbx new_executable_filename 1234
```

You can now see how useful it is to alter the child process code by adding a `sleep()` or similar command to trace both `fork()` and `exec()` calls.

8. The `dbx` for the child process should do a `detach` if it wishes to allow the child process to continue executing with no interference from the debugger. Alternately, a `kill` command should be used to terminate the process. If neither of these commands is used and a `dbx` quit command is used, the child process will be left in a suspended state.

```
demo% dbx a.out
Reading symbolic information...
Read 42 symbols
(dbx) list 1,15
    1   #include <stdio.h>
    2   main()
    3   {
    4           int pid;
    5
    6           pid = fork();
    7           printf("pid is %d 0, pid);
    8           switch(pid)
    9           {
   10           case -1:
   11                   perror("fork");
   12           case 0:
   13                   sleep(20);
   14           }
   15   }
(dbx) stop at 14
(2) stop at "child.c":15
(dbx) run
Running: a.out
pid is 0
pid is 1537
stopped in main at line 15 in file "child.c"
   15   }
(dbx)
```

In another commandtool or shelltool use the pid and read in the child process as shown in the following example (1537 is the pid of the sample process):

```
demo% dbx a.out 1537
Reading symbolic information...
Read 42 symbols
(dbx) list
   13            sleep(20);
   14        }
   15    }
   16
(dbx)
```

4.14. dbx FPA Support

1. The `fpaasm` debugger variable controls disassembly of FPA instructions. This variable may be set or displayed by means of the `dbxenv` command. The syntax of the command is:

```
dbxenv fpaasm <on|off>
```

If the value of `fpaasm` is `off`, all FPA instructions are disassembled as move instructions. If the value is `on`, FPA instructions are disassembled with FPA assembler mnemonics. Defaults: on a machine with an FPA, `fpaasm` is initially set to `on`; on machines without an FPA, it is initially set to `off`.

2. The `fpabase` debugger variable designates a 68020 address register for FPA instructions that use base-plus-short-displacement addressing to address the FPA. The syntax is:

```
dbxenv fpabase <a[0-7]|off>
```

If FPA disassembly is disabled (if `fpaasm` is `off`), its value is ignored. Otherwise, its value is interpreted as follows:

value in `[a0..a7]`:
Long move instructions that use the designated address register in base-plus-short-displacement mode are assumed to address the FPA, and are disassembled using FPA assembler mnemonics. Note that this is independent of the actual run-time value of the register.

value = `off`:
All based-mode FPA instructions are disassembled and single-stepped as move instructions.

The default value of `fpabase` is `off`, which designates no FPA base register.

4.15. Example of FPA Disassembly

Consider the following simple FORTRAN program:

```
program example
print *,f(1.0,1.0)
end

function f(x,y)
f = atan(x/y)
return
end
```

Assume that this program has been compiled with the `-g` option into the file `example`. On a Sun-3 with an FPA, we could disassemble the function `f` as shown below. Note that the FORTRAN intrinsic `ATAN` is directly supported by the FPA instruction set and the FORTRAN compiler.

```
% dbx a.out
(dbx) stop in f
(1) stop in f
(dbx) run
Running: a.out
stopped in f at line 5 in file "example.f"
    5           f = atan(x/y)
(dbx) &$pc/8i
f+0x12:         movl    a6@(0xc),a0
f+0x16:         fpmoves a0@,fpa0
f+0x1c:         movl    a6@(0x8),a0
f+0x20:         fprdivs a0@,fpa0
f+0x26:         fpmoves fpa0,a6@(-0xc)
f+0x2e:         fpmoves a6@(-0xc),fpa1
f+0x36:         fpatans fpa1,fpa1
f+0x40:         fpmoves fpa1,a6@(-0x8)
    ...
```

FPA disassembly can be disabled by setting the debugger variable `fpaasm` to `off`. This causes `dbx` to disassemble FPA instructions as long moves to addresses on the FPA page:

```
(dbx) dbxenv fpaasm off
(dbx) &f+0x12/10i
f+0x12:         movl    a6@(0xc),a0
f+0x16:         movl    a0@,0xe0000c00:l
f+0x1c:         movl    a6@(0x8),a0
f+0x20:         movl    a0@,0xe0000600:l
f+0x26:         movl    0xe0000e00:l,a6@(-0xc)
f+0x2e:         movl    a6@(-0xc),0xe0000c08:l
f+0x36:         movl    #0x41,0xe0000818:l
f+0x40:         movl    0xe0000e08:l,a6@(-0x8)
```

When tracing a more complex program, one may occasionally want to step into a routine that has been compiled with optimization on. In such routines, it is often the case that the compiled code addresses the FPA page by using base+short offset addressing. Such code can be difficult to recognize unless it is known ahead of time that a particular address register is being used to address the FPA. This situation can be identified by the presence of an instruction that loads the address of the FPA page (0xe0000000) into an address register before doing any floating-point arithmetic.

For example, here is a disassembly of the beginning of an optimized FORTRAN routine compiled with the `-O` and `-ffpa` options:

```
(dbx) &ddot_/7i
ddot_:        link     a6,#-0x2a0
ddot_+0x4:    moveml   #<d2,d3,d4,d5,d6,d7,a2,a3,a4,a5>,sp@
ddot_+0x8:    lea      e0000000:l,a2
ddot_+0xe:    movl     a2@(0xe20),a6@(-0x278)
ddot_+0x14:   movl     a2@(0xe24),a6@(-0x274)
ddot_+0x1a:   movl     a2@(0xe28),a6@(-0x270)
ddot_+0x20:   movl     a2@(0xe2c),a6@(-0x26c)
```

`dbx` does not know which register (if any) is being used to address the FPA in a given sequence of machine code. However, you may set the `dbxenv` variable `fpabase` to designate an MC68020 address register as an FPA base register. In this example, we note that the compiler has loaded the address of the FPA page into register `a2`, and so we designate `a2` as the FPA base register to obtain the following:

```
(dbx) dbxenv fpabase a2
(dbx) &ddot_/7i
ddot_:        link     a6,#-0x2a0
ddot_+0x4:    moveml   #<d2,d3,d4,d5,d6,d7,a2,a3,a4,a5>,sp@
ddot_+0x8:    lea      e0000000:l,a2
ddot_+0xe:    fpmoved@2        fpa4,a6@(-0x278)
ddot_+0x1a:   fpmoved@2        fpa5,a6@(-0x270)
ddot_+0x26:   fpmoved@2        204ce:l,fpa5
ddot_+0x36:   fpmoved@2        204ce:l,fpa4
```

4.16. Examples of FPA Register Use

FPA data registers can be displayed using a syntax similar to that used for the MC68881 co-processor registers. Note that unlike the MC68881 registers, FPA registers may contain either single-precision (32-bit) or double-precision (64-bit) values; MC68881 registers always contain an extended-precision (96-bit) value.

For example, if `fpa0` contains the single-precision value 2.718282, we may display it as follows:

```
(dbx) &$fpa0/f
fpa0       0x402df855    +2.718282e+00
```

Note that the value is displayed in hexadecimal as well as in floating- point notation.

A double-precision value may be displayed using the `/F` format. For example, if `fpa0` contains the double-precision value 2.718281828, we may display it as follows:

```
(dbx) &$fpa0/F
fpa0      0x4005bf0a 0x8b04919b     +2.71828182800000e+00
```

Note that it is important to use the correct display format; attempting to display a double-precision value in single precision (and vice versa) will usually produce meaningless results.

FPA registers can also be used in `set` commands and in arithmetic expressions. Since `dbx` cannot tell whether the value in an FPA register is single or double precision, `dbx` provides two sets of names to refer to FPA registers. The names `{$fpa0..$fpa31}` always cause the contents of the register to be interpreted as a double-precision value; the names `{$fpa0s..$fpa31s}` cause interpretation as a single-precision value. Thus, the commands

```
(dbx) set $fpa0s = 1.0
(dbx) set $fpa0 = 1.0
```

cause different bit patterns to be stored in `fpa0`.

5

Debugging Tips for Programmers

This chapter provides a number of debugging tips. Primarily, the examples presented here are in the FORTRAN language. However, with some minor changes and modifications, the sample program and the examples in this chapter may also be of use to C language programmers.

NOTE *FORTRAN arrays can be specified using either parentheses* () *or brackets* []. `dbx` *can take both.*

Sample program

The following sample program (with bug) is used in several examples:

a1.f

```
      parameter ( n=2 )
      real twobytwo(2,2) / 4 *-1 /
      call mkidentity( twobytwo, n )
      print *, determinant( twobytwo )
      end
```

a2.f

```
      subroutine mkidentity ( array, m )
      real array(m,m)
      do 10 i = 1, m
      do 20 j = 1, m
      if ( i .eq. j ) then
          array(i,j) = 1.
      else
          array(i,j) = 0.
      endif
   20 continue
   10 continue
      return
      end
```

a3.f

```
      real function determinant ( a )
      real a(2,2)
      determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
      return
      end
```

5.1. dbx and FORTRAN

Note the following when using dbx with FORTRAN programs:

1) The main routine is referenced as MAIN (as distinguished from main). All other names in the source file that have upper case letters in them will be lower case in dbx, unless the program was compiled with f77 -U.

2) When referring to the value of a logical type in an expression, use the value 0 or 1 rather than .false. or .true., respectively.

5.2. A Sample dbx Session

A few dbx commands are shown here in examples, using the sample program at the start of this chapter.

Throughout a debugging session, dbx defines a procedure and a source file as *current*. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, "**stop at 5**" sets one of three different breakpoints depending on whether the current file is a1.f, a2.f, or a3.f.

compile

To use dbx or dbxtool, you must compile and load your program with the -g flag.* For example:

```
demo% f77 -o silly -g a1.f a2.f a3.f
```

or:

```
demo% f77 -c -g a1.f a2.f a3.f
demo% f77 -g -o silly a1.o a2.o a3.o
```

run

To run the program under the control of dbx, change to the directory where the sources and programs reside, then type the dbx command and the name of the executable file:

```
demo% dbx silly
Reading symbolic information...
Read 307 symbols
(dbx)
```

quit

* The -g and -O options are incompatible. If used together, the -g option cancels the -O option.

To quit dbx, enter the command **quit**.

breakpoint

To set a breakpoint before the first executable statement, wait for the (dbx) prompt, then type "**stop in MAIN**".

```
(dbx) stop in MAIN
(2) stop in MAIN
(dbx)
```

run

After the (dbx) prompt appears, type **run** to begin execution. When the breakpoint is reached, dbx displays a message showing where it stopped, in this case at line 3 of file a1.f.

```
(dbx) run
Running: silly
stopped in MAIN at line 3 in file "a1.f"
    3           call mkidentity( twobytwo, n )
(dbx)
```

print

The command "**print n**" displays 2, since dbx knows about parameters.

```
(dbx) print n
n = 2
(dbx)
```

The command "**print twobytwo**" displays the entire matrix, one element per line. Note that dbx displays square brackets (not parentheses) when it references array elements.

```
(dbx) print twobytwo
twobytwo = [1,1]        -1.0
        [2,1]   -1.0
        [1,2]   -1.0
        [2,2]   -1.0

(dbx)
```

The command "**print array**" fails because mkidentity is not active at this point.

```
(dbx) print array
"array" is not active
(dbx)
```

next

The command **next** advances execution to line 4, and if the command "**print twobytwo**" is now repeated, it displays the unit matrix.

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4           print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo = [1,1]        1.0
        [2,1]   0.0
        [1,2]   0.0
        [2,2]   1.0

(dbx) quit
demo%
```

Calling a Function

It is possible to call a subroutine or function in the program at any point when execution has stopped. The effect is exactly as if the source had contained a call at that point. For example if, after the initial "**stop in MAIN**" described above, you typed "**print determinant(twobytwo)**", dbx displays the value 0.0, since mkidentity would not yet have modified twobytwo.

```
demo% dbx silly
Reading symbolic information...
Read 283 symbols
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) print determinant(twobytwo)
determinant(twobytwo) = 0.0
(dbx)
```

This facility is often useful for special-case printing. For example, in a program it might be meaningful to trace the row and column sums of different matrices. A subroutine called matsum that does this could be compiled into a program and invoked by the user at appropriate breakpoints.

Structures and Pointers

The dbx debugger recognizes the Sun FORTRAN items such as *structure*, *record*, *union*, and *pointer*. The following examples show using dbx with these items.

Compile for dbx using the **-g** option, load it in dbx, and list it.

```
demo% f77 -o debstr -g deb1.f
deb1.f:
 MAIN:
demo% dbx debstr
Reading symbolic information...
Read 269 symbols
(dbx) list 1,30
     1   * deb1.f: Show dbx with structures and pointers
     2         STRUCTURE  /PRODUCT/
     3                 INTEGER*4       ID
     4                 CHARACTER*16    NAME
     5                 CHARACTER*8     MODEL
     6                 REAL*4          COST
     7                 REAL*4          PRICE
     8         END STRUCTURE
     9
    10         RECORD /PRODUCT/ PROD1, PROD2
    11         POINTER (PRIOR, PROD1), (CURR, PROD2)
    12
    13         PRIOR = MALLOC( 36 )
    14         PROD1.ID = 82
    15         PROD1.NAME = "Schlepper"
    16         PROD1.MODEL = "XL"
    17         PROD1.COST = 24.0
    18         PROD1.PRICE = 104.0
    19         CURR = MALLOC( 36 )
    20         PROD2 = PROD1
    21         WRITE ( *, * ) PROD2.NAME
    22         STOP
    23         END
(dbx)
```

Set a breakpoint at a specific line number, and run it under dbx.

```
(dbx) stop at 21
(1) stop at "deb1.f":21
(dbx) run
Running: debstr
stopped in main at line 21 in file "deb1.f"
   21          WRITE ( *, * ) PROD2.NAME
(dbx)
```

Print and inquire about a record.

```
(dbx) print prod1
*prod1 = (
        id    =         82
        name  =         "Schlepper       "
        model =         "XL      "
        cost  = 24.0
        price = 104.0
)
(dbx) whatis prod1
 (based variable) structure /product/ prod1
(dbx)
```

If you tell dbx to print a record, it displays all fields of the record, including field names.

Print a pointer, then quit dbx.

```
(dbx) print prior
prior = 166868
(dbx) quit
demo%
```

If you tell it to print a pointer, it displays the contents of that pointer, which is the address of the variable pointed to. This address could very well be different with every run.

Parameters

The dbx debugger recognizes parameters — the compiler generates pseudo variables for parameters when programs are compiled for dbx with the **-g** option. The following examples show using dbx with parameters.

Compile for dbx using the **-g** option, load it in dbx and list it. Print some parameters.

```
demo% f77 -o silly -g deb2.f a2.f a3.f
deb2.f:
deb2.f:
 MAIN silly:
a2.f:
a2.f:
        mkidentity:
a3.f:
a3.f:
        determinant:
Linking:
demo% dbx silly
Reading symbolic information...
Read 269 symbols
(dbx) list 1,30
    1           program silly
    2           parameter ( n=2, nn=n*n )
    3           real twobytwo(n,n)
    4           data twobytwo /nn *-1 /
    5           call mkidentity( twobytwo, n )
    6           print *, determinant(twobytwo)
    7           end
(dbx) print n
'deb2'MAIN'n = 2
(dbx) print nn
nn = 4
(dbx) quit
demo%
```

Uppercase

If your program has uppercase letters in any identifiers, and you want dbxtool to recognize them, then you need to give dbxtool a specific command, as follows.

```
dbxenv case insensitive
```

Once you've done the above command, then when dbxtool finds and displays uppercase identifiers, you can select them and dbxtool can find them.

Caveat: Once you've done the above command, then the command "**stop in MAIN**" does not work.

Parts of Large Arrays

Printing portions of large arrays is often of interest to FORTRAN programmers. For example:

```
demo% dbx a.out
Reading symbolic information...
Read 314 symbols
(dbx) list 1,25
    1           integer *4 i(5,5)
    2           do 10 j = 1,5
    3             do 20 k = 1,5
    4               i(j,k) = (j * 10) + k
    5      20     continue
    6      10   continue
    7           stop
    8           end
(dbx) stop at 7
(1) stop at "temp.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "temp.f"
    7           stop
(dbx) &i(2,3)/12D
0x000214b0:  23 33 43 53
0x000214c0:  14 24 34 44
0x000214d0:  54 15 25 35
(dbx)
```

Note that the **D** in the last `dbx` command shown in the above example is the mode used to display a longword in decimal format.

Passing Arguments to a Main Program

To specify main program arguments correctly within dbx, place them on the run command of dbx, as follows:

Note that the arguments are passed *not* on the dbx or dbxtool command line, nor on the debug command line.

```
demo% cat tesargs.f
    character argv*10
    integer i, iargc, m
    m = iargc()
    i = 1
    do while  ( i .le. m )
        call getarg ( i, argv )
        write( *, '( i2, 1x, a )' ) i, argv
        i = i + 1
    end do
    stop
    end
demo % a.out first second last
 1 first
 2 second
 3 last

demo% dbx a.out
Reading symbolic information...
Read 292 symbols
(dbx) run first second last
Running: a.out first second last
 1 first
 2 second
 3 last
execution completed, exit code is 0
program exited with 0
(dbx)
```

Where Exception Occurred

You can find the source code line where a floating-point exception occurred by using the ieee_handler routine with either dbx or dbxtool. For example:

Note the "catch FPE" dbx command. →

```
demo% cat divide.f
    external myhandler
    ieeer = ieee_handler('set', 'all', myhandler)
    r = 14.2
    s = 0.0
    print *,r/s
    stop
    end

    integer function myhandler(sig, code, context)
    integer sig, code, context(5)
    call abort()
    end

demo% f77 -g -f68881 divide.f
divide.f:
 MAIN:
    myhandler:
demo% dbx a.out
Reading symbolic information...
Read 233 symbols
(dbx) catch FPE
(dbx) run
Running: a.out
signal FPE (floating point exception)
    in MAIN at line 5 in file "divide.f"
    5            print *,r/s
(dbx) quit
demo%
```

Print in Hex

Although you cannot use the `print` command to display objects in hexadecimal, you can use the `alias` command with machine-level commands to achieve the same results. The following command creates a new command named `mem` which requires one argument: an object of type `integer*4`. It then displays that argument in hexadecimal. For comparison, the example below shows the same value displayed in decimal using the `print` command. The "`1`" is the number of words to print.

```
(dbxtool) alias mem "print (void *) (!:1)"
(dbxtool) mem i(2,4)
(void *) i[2,4] = 0x18
(dbxtool) print i(2,4)
i[2,4] = 24
(dbxtool) quit
demo%
```

Using the following command, you can now set up a button in dbxtool so that the mouse could select the object.

```
(dbxtool) button expand mem
```

5.3. Using adb with FORTRAN

This section introduces the use of the adb low-level debugger with the FORTRAN language.

The adb debugger can be used to provide a stack traceback at a lower level. adb can be used on any program regardless of whether or not it was compiled with the **-g** debugging flag. For more information on adb, see adb Tutorial, Chapter 6.

The adb program does *not* display any prompt at all; it just waits for input; except if you enter only a Return, then it will display the prompt adb.

compile

With the same three files as in the first dbx example, if you compile and run, you get NaN (not a number). If you get an abort, you can get an adb low-level traceback; so force an abort with an exception handler.

revised a1.f

```
        parameter ( n=2 )
        real twobytwo(2,2) / 4 *-1 /
        external hand
        i = ieee_handler ( 'set', 'all', hand )
        call mkidentity( twobytwo, n )
        print *, determinant( twobytwo )
        end

        integer function hand ( sig, code, context )
        integer sig, code, context(5)
        call abort()
        end
```

Here is a compile and run, for a Sun-3, with 68881 floating point.

```
demo% f77 -f68881 -o silly a1.f a2.f a3.f
a1.f:
a1.f:
 MAIN:
        hand:
a2.f:
a2.f:
        mkidentity:
a3.f:
a3.f:
        determinant:
Linking:
demo% silly
abort: called
Abort (core dumped)
demo%
```

start

You can start up adb and display a C backtrace as follows.

```
demo% adb silly core
core file = core -- program ''silly''
SIGIOT 6: abort
$c
_kill(?)
__DYNAMIC() + 6
_force_abort() + 1c
_abort_() + 4a
_hand_() + 18
_hand_(?)
_determinant_(0x20258) + 18
_MAIN_() + 6e
_main(0x1,0xefffd8c,0xefffd94) + 5a
```

Interpretation (bottom up):

- The startup routine main, called the FORTRAN MAIN routine,
- which in turn called the function determinant,
- which in turn called the function hand,
- which in turn called the function abort,
- which in turn called the function force_abort to halt execution.

instructions

Display, say, 10(hex) machine instructions and their addresses starting from the entry point determinant.

```
determinant_,10?ia
_determinant_:
_determinant_:          linkw   a6,#0
_determinant_+4:                          addl      #-8,a7
_determinant_+0xa:                        moveml    #0,sp@
_determinant_+0xe:                        fmovemx   ,a6@(-8:1)
_determinant_+0x18:                       movl      a6@(8),a0
_determinant_+0x1c:                       movl      a6@(8),a1
_determinant_+0x20:                       fmoves    a1@,fp0
_determinant_+0x24:                       fmuls     a0@(0xc),fp0
_determinant_+0x2a:                       movl      a6@(8),a0
_determinant_+0x2e:                       fmoves    a0@(8),fp1
_determinant_+0x34:                       movl      a6@(8),a0
_determinant_+0x38:                       fdivs     a0@(4),fp1
_determinant_+0x3e:                       fsubx     fp1,fp0
_determinant_+0x42:                       fmoves    fp0,a6@(-8)
_determinant_+0x48:                       nop
_determinant_+0x4a:                       movl      a6@(-8),d0
_determinant_+0x4e:
```

quit

To quit adb, type **$q** or **$Q** or **^D**. For example:

```
$q
demo%
```

blank common

Variables can be displayed in a variety of formats, but their addresses must be known. The addresses of some external variables are easy to determine.

For example, to print the first four bytes after the label __BLNK__, in a decimal format, do this.

```
__BLNK__/D
```

which is equivalent to the dbx command "**print n**" if n is the first variable in blank common.

The addresses of local variables are usually difficult to determine.

unformatted files

As another example, consider this program.

```
      write(4) 4
      end
```

When executed, this program creates a file named `fort.4` which contains a single unformatted record. An unformatted record includes two count words containing the record length at the beginning and end of the record.

You can examine this data file with `adb` as follows.

```
demo% adb fort.4 -
```

Then display the first three words of the data file (start at location 0, for 3 times, in decimal format).

```
0,3?D
0:              4               4               4
$q
demo%
```

6

adb Tutorial

6.1. A Quick Survey

Available on most UNIX systems, adb is a debugger that permits you to examine core files resulting from aborted programs, display output in a variety of formats, patch files, and run programs with embedded breakpoints. This chapter provides examples of the most useful features of adb. The reader is expected to be familiar with basic SunOS commands, and with the C language.

NOTE *This chapter describes* adb *use on the Sun-3 and Sun-4 only. Chapter 7 describes* adb *use on the Sun386i.*

Starting adb

Start adb with a shell command of the form

```
% adb [objectfile] [corefile]]
```

where *objectfile* is an executable SunOS file and *corefile* is a core dump file. If the object file is named a.out, then the invocation is

```
% adb
```

If you place object files into a named *program* file, then the invocation is

```
% adb program
```

The filename minus (–) means ignore the argument, as in:

```
% adb - core
```

This is for examining the core file without reference to an object file. adb provides requests for examining locations in either file: **?** examines the contents of *objectfile*, while **/** examines the contents of *corefile*. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

Current Address

adb maintains a current address, called **dot**. When an address is entered, the current address is set to that location, so that

```
0126?i
```

sets dot to octal 126 and displays the instruction at that address. The request

```
.,10/d
```

displays 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item displayed. When used with the **?** or **/** requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up of decimal integers, octal integers, hexadecimal integers, and symbols from the program under test. These may be combined with the operators **+** (plus), **-** (minus), ***** (multiply), **%** (integer divide), **&** (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, you can type name. On a Sun-3 or Sun-4 you could alternatively type _name; adb recognizes both forms on these systems.

Formats

To display data, specify a collection of letters and characters to describe the format of the display. Formats are remembered, in the sense that typing a request without a format displays the new output in the previous format. Here are the most commonly used format letters:

Table 6-1 *Some* adb *Format Letters*

Letter	*Description*
b	one byte in octal
B	one byte in hex
c	one byte as a character
o	one 16-bit word in octal
d	one 16-bit word in decimal
f	one single-precision floating point value
i	MC68020 instructions on Sun-3, SPARC instruction on Sun-4.
s	a null-terminated character string
a	the value of dot
u	one 16-bit word as an unsigned integer
n	print a newline
r	print a blank space
^	backup dot (not really a format)
+	advance dot (not really a format)

Format letters are also available for `long` values: for example, `D` for long decimal, and `F` for double-precision floating point. Since integers are long words on the Sun-3 capital letters are used more often than not.

General Command Meanings

The general form of a command is:

```
[address[,count]] command [modifier]
```

which sets dot to *address* and executes *command count* times. The following table illustrates some general adb command meanings:

Table 6-2 *Some* adb *Commands*

Some* adb *Commands	
Command	*Meaning*
?	Print contents from object file
/	Print contents from core file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

Since adb catches signals, 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.

6.2. Debugging C Programs

If you use adb because you are accustomed to it, you will want to compile programs with the -go option, to produce old-style symbol tables. This will make debugging proceed according to expectations. If you don't compile programs with -go and the -O option is set, the object code will be optimized, and may not so readily be understood as the same thing that was written in the source file.

Debugging A Core Image

Consider the C program below, which illustrates a common error made by C programmers. The object of the program is to change the lower case t to an upper case T in the string pointed to by ch, and then write the character string to the file indicated by the first argument.

```
#include <stdio.h>

char *cp = "this is a sentence.";

main(argc, argv)
int argc;
char **argv;
{
    FILE *fp;
    char c;

    if (argc == 1) {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }
    if ((fp = fopen(argv[1], "w")) == NULL) {
        perror(argv[1]);
        exit(2);
    }
    cp = 'T';
    while (c = *cp++)
        putc(c, fp);
    fclose(fp);
    exit(0);
}
```

The bug is that the character T is stored in the pointer cp instead of in the string pointed to by cp. Compile the program as follows:

```
% cc -go example1.c
% a.out junk
Segmentation fault (core dumped)
```

Executing the program produces a core dump caused by an illegal memory reference. Now invoke adb by typing:

```
% adb
core file = core -- program "a.out"
memory fault
```

Commonly the first debugging request given is

```
$c
_main[8074](2,fffd7c,fffd88) + 92
```

which produces a C backtrace through the subroutines called. The output from adb tells us that only one function — main — was called, and the arguments argc and argv have the hexadecimal values 2 and fffd7c, respectively. Both these values look reasonable — 2 indicates two arguments, and fffd7c is the stack address of the parameter vector. The next request

```
$C
_main[8074](2,fffd7c,fffd88) + 92
    fp:         10468
    c:          104
```

generates a C backtrace plus an interpretation of all the local variables in each function, and their values in hexadecimal. The value of the variable c looks incorrect since it is outside the ASCII range. The request

```
$r
d0    54        frame+24
d1    77        frame+47
d2    2         man1
d3    0         exp
d4    0         exp
d5    0         exp
d6    0         exp
d7    0         exp
a0    54        frame+24
a1    0         exp
a2    0         exp
a3    fffd7c
a4    fffd88
a5    0         exp
a6    fffd64
sp    fffd5c
pc    8106      _main+92
ps    0         exp
_main+92:               ???
```

displays the registers, including the program counter, and an interpretation of the instruction at that location. The request

```
$e
_environ:   fffd88
_sys_nerr:  48
__ctype_:   202020
__exit_nhandlers:       0
__exit_tnames:          9b06
__lastbuf:  10684
__root:     0
```

```
__lbound:    0
__ubound:    0
curbrk:      12dd4
__d_pot:     8000
__d_big_pot:             8000
__d_r_pot:   8000
__d_r_big_pot:           8000
_errno:      0
_end:        0
```

displays the values of all external variables.

A map exists for each file handled by adb. The map for object files is referenced by ?, whereas the map for core files is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To display information about maps, type:

```
$m
b1 = 2000            e1 = b000            f1 = 800
b2 = 10000           e2 = 11000           f2 = 3800
/ map        'core'
b1 = 10000           e1 = 13000           f1 = 1800
b2 = fff000          e2 = 1000000         f2 = 4800
```

This produces a report of the contents of the maps. More about these maps later.

In our example, we might want to see the contents of the string pointed to by cp. We would want to see the string pointed to by cp in the core file:

```
*cp/s
55:
data address not found
```

Because the pointer was set to 'T' (hex 54) and then incremented, it now equals hex 55. On the Sun-3, there are no symbols below address 2000 (8000 on a Sun-2), so the data address 55 cannot be found. We could also display information about the arguments to a function. To get the decimal value of the argc argument to main, which is a long integer, type:

```
main.argc/D
fffd6c:        2
```

To display the hex values of the three consecutive cells pointed to by argv in the function main, type:

```
*main.argv,3/X
fffd7c:        fffdc0         fffdc6         0
```

Note that these values are the addresses of the arguments to main. Therefore, typing these hex values should yield the command-line arguments:

```
fffdc0/s
fffdc0:         a.out
```

The request

```
.=
                fffdc0
```

displays the current address (not its contents) in hex, which has been set to the address of the first argument. The current address, dot, is used by `adb` to remember its current location. It allows the user to reference locations relative to the current address. For example

```
fffdc6:         zzz
```

prints the first command-line argument.

Setting Breakpoints

Set breakpoints in a program with the `:b` instruction, which has this form:

```
address:b  [ request ]
```

Consider the C program below, which changes tabs into blanks, and is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

```
#include <stdio.h>

#define MAXLIN  80
#define YES 1
#define NO  0
#define TABSP   8

int tabs[MAXLIN];

main()
{
    int *ptab, col, c;

    ptab = tabs;
    settab(ptab);   /* set initial tab stops */
    col = 1;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '\t':
            while (tabpos(col) != YES) {
                putchar(' ');
                col++;
            }
            putchar(' ');
            col++;
```

```
            break;
        case '\n':
            putchar('\n');
            col = 1;
            break;
        default:
            putchar(c);
            col++;
        }
    }
    exit(0);
}

tabpos(col) /* return YES if col is a tab stop, NO if not */
int col;
{
    if (col > MAXLIN)
        return(YES);
    else
        return(tabs[col]);
}

settab(tabp)    /* set initial tab stops every TABSP spaces */
int *tabp;
{
    int i;

    for (i = 0; i <= MAXLIN; i++)
        (i % TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Run the program under the control of adb, and then set four breakpoints as follows:

```
% adb a.out -
settab:b
tabpos:b
```

This sets breakpoints at the start of the two functions. Sun compilers generate statement labels only with the –g option, which is incompatible with adb. Therefore it is impossible to plant breakpoints at locations other than function entry points using adb. To display the location of breakpoints, type:

```
$b
breakpoints
count   bkpt            command
1       _tabpos
1       _settab
```

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 this example no command fields are present.

Display the instructions at the beginning of function `settab()` in order to observe that the breakpoint is set after the `link` assembly instruction:

```
settab,5?ia
_settab:
_settab:                    link    a6,#0
_settab:                    addl    #-4,a7
_settab+a:                  moveml  #<>,sp@
_settab+e:                  clrl    a6@(-4)
_settab+12:                 cmpl    #50,a6@(-4)
_settab+1a:
```

This request displays five instructions starting at `settab` with the address of each location displayed. Another variation is

```
settab,5?i
_settab:
_settab:                    link    a6,#0
                            addl    #-4,a7
                            moveml  #<>,sp@
                            clrl    a6@(-4)
                            cmpl    #50,a6@(-4)
```

which displays the instructions with only the starting address. Note that we accessed the addresses from `a.out` with the **?** command. In general, when asking for a display of multiple items, adb advances the current address the number of bytes necessary to satisfy the request; in the above example, five instructions were displayed and the current address was advanced 26 bytes.

To run the program, type:

```
:r
```

To delete a breakpoint, for instance the entry to the function `tabpos()`, type:

```
tabpos:d
```

Once the program has stopped, in this case at the breakpoint for `settab()`, adb requests can be used to display the contents of memory. To display a stack trace, for example, type:

```
$c
_settab[8250](10658) + 4
_main[8074](1,fffd84,fffd8c) + 1a
```

And to display three lines of eight locations each from the array called `tabs`, type:

```
tabs,3/8X
_tabs:
_tabs:      0    0    0    0    0    0    0    0
            0    0    0    0    0    0    0    0
            0    0    0    0    0    0    0    0
```

At this time (at location `settab`) the `tabs` array has not yet been initialized. If you just deleted the breakpoint at `tabpos`, put it back by typing:

```
tabpos:b
```

To continue execution of the program from the breakpoint type:

```
:c
    x
```

You will need to give the `a.out` program a line of data, as in the figure above. Once you do, it will encounter a breakpoint at `tabpos+4` and stop again. Examine the `tabs` array once more: now it is initialized, and has a one set in every eighth location:

```
tabs,3/8X
_tabs:
_tabs:      1    0    0    0    0    0    0    0
            1    0    0    0    0    0    0    0
            1    0    0    0    0    0    0    0
```

You will have to type `:c` eight more times in order to get your line of output, since there is a breakpoint at every input character. Type CTRL-D to terminate the running process and to return to the command level of `adb`.

Advanced Breakpoint Usage

The 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 passed on to the test program if you type:

```
:c 0
```

Now let's reset the breakpoint at `settab()` and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b settab,5?ia
:r
_settab:
_settab:                 link     a6,#0
_settab+4:               addl     #-4,a7
_settab+a:               moveml   #<>,sp@
_settab+e:               clrl     a6@(-4)
_settab+12:              cmpl     #50,a6@(-4)
_settab+1a:
breakpoint      _settab+4:               addl     #-4,a7
```

It is possible to stop every two breakpoints, if you type ,2 before the breakpoint command. Variables can also be displayed at the breakpoint, as illustrated below.

```
tabpos+4,2:b main.col?X
:c
    x
fffd64:         1
fffd64:         2
breakpoint      _tabpos+4:               addl     #0,a7
```

This shows that the local variable `col` changes from 1 to 2 before the occurrence of the breakpoint.

NOTE *Setting a breakpoint causes the value of dot to be changed. However, executing the program under* adb *does not change the value of dot.*

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b main.ptab/X; main.c/X
:r
fffd68:         10658
fffd60:         0
breakpoint      _settab+4:               addl     #-4,a7
```

A semicolon is used to separate multiple `adb` requests on a single line.

Other Breakpoint Facilities

Arguments and redirection of standard input and output are passed to a program as follows. This request kills any existing program under test and starts the object file anew:

```
:r arg1 arg2 ... <infile >outfile
```

The program being debugged can be single stepped as follows. If necessary, this request starts up the program being debugged and stops after executing the first instruction:

```
:s
```

You can enter a program at a specific address by typing:

```
address:r
```

The count field can be used to skip the first *n* breakpoints, as follows:

```
,n:r
```

This request may also be used for skipping the first *n* breakpoints when continuing a program:

```
,n:c
```

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
```

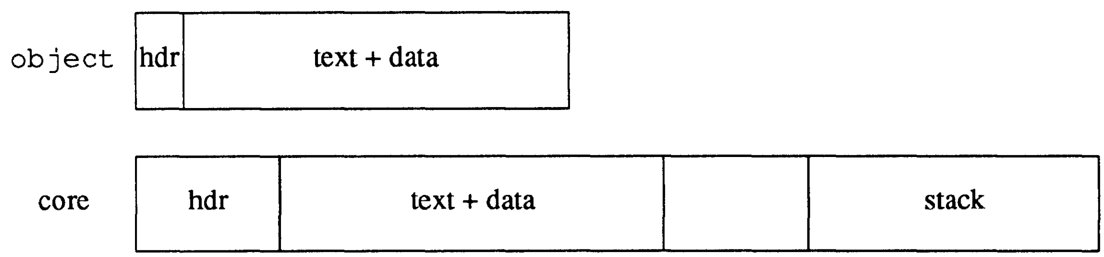
6.3. File Maps

SunOS supports several executable file formats. Executable type 407 is generated by the cc (or ld) flag –N. Executable type 410 is generated by the flag –n. An executable type 413 is generated by the flag –z; the default is type 413. adb interprets these different file formats, and provides access to the different segments contained in them through a set of maps. To display the maps, type **$m** inside adb.

407 Executable Files

In 407-format files, instructions and data are intermixed. This makes it impossible for adb to differentiate data from instructions, but adb will display in either format. Furthermore, some displayed symbolic addresses look incorrect (for example, data addresses as offsets from routines). Here is a picture of 407-format files:

Figure 6-1 *Executable File Type 407*



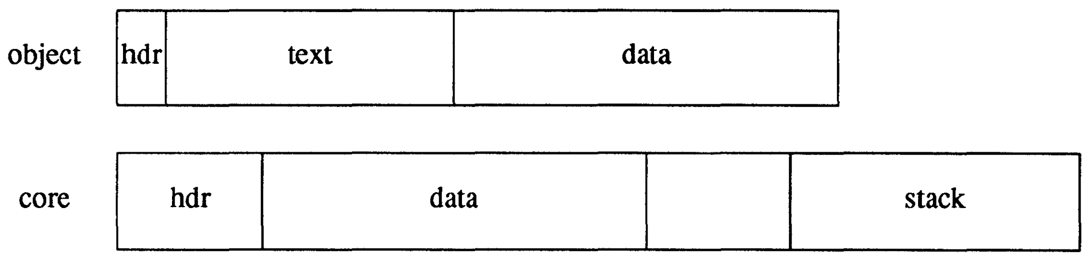
Here are the maps and variables for 407-format files:

```
$m
? map         'object'
b1 = 2000                e1 = 8f28              f1 = 20
b2 = 8000                e2 = 9560              f2 = 20
/ map         'core'
b1 = 8000                e1 = b800              f1 = 1800
b2 = fff000              e2 = 1000000           f2 = 5000
$v
variables
b = 0100000
d = 03070
e = 0407
m = 0407
s = 010000
t = 07450
```

410 Executable Files

In 410-format files (pure executable), instructions are separate from data. The `?` command accesses the data part of the object file, telling `adb` to use the second part of the map in that 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. Here is a picture of 410-format files:

Figure 6-2 *Executable File Type 410*



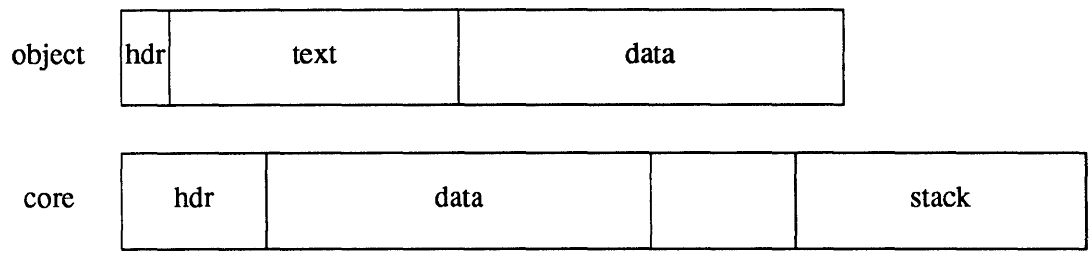
Here are the maps and variables for 410-format files:

```
$m
? map        'object'
b1 = 2000             e1 = 8f28               f1 = 20
b2 = 10000            e2 = 10638              f2 = f48
/ map        'core'
b1 = 10000            e1 = 12800              f1 = 1800
b2 = fff000           e2 = 1000000            f2 = 4000
$v
variables
b = 0200000
d = 03070
e = 0410
m = 0410
s = 010000
t = 07450
```

413 Executable Files

In 413-format files (pure demand-paged executable) the instructions and data are also separate. However, in this case, since data is contained in separate pages, the base of the data segment is also relative to address zero. In this case, since the addresses overlap, it is necessary to use the `?*` operator to access the data space of the object file. In both 410 and 413-format files the corresponding core file does not contain the program text. Here is a picture of 413-format files:

Figure 6-3 *Executable File Type 413*



The only difference between a 410 and a 413-format file is that 413-format segments are rounded up to page boundaries. Here are the maps and variables for 413-format files:

```
$m
? map        'abort'
b1 = 2000                  e1 = 9000                  f1 = 800
b2 = 10000                 e2 = 10800                 f2 = 1800
/ map        'core'
b1 = 10000                 e1 = 12800                 f1 = 1800
b2 = fff000                e2 = 1000000               f2 = 4000
$v
variables
b = 0200000
d = 04000
e = 0413
m = 0413
s = 010000
t = 010000
```

Variables

The `b`, `e`, and `f` fields are used to map addresses into file addresses. The `f1` field is the length of the header at the beginning of the file — 020 bytes for an object file and 02000 bytes for a core file. The `f2` field is the displacement from the beginning of the file to the data. For a 407-format file with mixed text and data, this is the same as the length of the header; for 410-format and 413-format files, this is the length of the header plus the size of the text portion. The `b` and `e` fields are the starting and ending locations for a segment. Given the address *A*, the location in the file (either object or core) is calculated as:

```
b1<A<e1    file address = (A-b1)+f1
b2<A<e2    file address = (A-b2)+f2
```

You can access locations by using the adb-defined variables. The $v request displays the variables initialized by adb:

b base address of data segment,

d length of the data segment,

s length of the stack,

t length of the text,

m execution type (407, 410, 413).

Those variables not presented are zero. Use can be made of these variables by expressions such as

```
<b
```

in the address field. Similarly, the value of a variable can be changed by an assignment request such as

```
02000>b
```

which sets b to octal 2000. These variables are useful to know if the file under examination is an executable or core image file.

The adb program 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.

6.4. Advanced Usage

One of the uses of adb is to examine object files without symbol tables since dbx cannot handle this kind of task.

With adb, you can combine formatting requests to provide elaborate displays. Several examples are given below.

Formatted Dump

The following adb command line displays four octal words followed by their ASCII interpretation from the data space of the core file:

```
<b,-1/4o4^8Cn
```

Broken down, the various requests 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^` Back up 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 displayed as followed by the corresponding character in the range 0140 to 0177. An `@` is displayed as `@@`.

`n` Print a newline.

The following request could have been used instead to allow the displaying to stop at the end of the data segment. (The request `<d` provides the data segment size in bytes.)

```
<b?<d/404^8Cn
```

Because `adb` can read in scripts, you can use formatting requests to produce image dump scripts. Invoke `adb` as follows:

```
% adb objectfile corefile < dumpfile
```

This reads in a script file, `dumpfile`, containing formatting requests. Here is an example of such a script:

```
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,-1/8ona
```

The request `120$w` sets the width of the output to 120 characters (normally, the width is 80 characters). `adb` attempts to display addresses as:

```
symbol + offset
```

The request `4095$s` increases the maximum permissible offset to the nearest symbolic address from the default 255 to 4095. The request = can be used to display literal strings. Thus, headings are provided in this dump program with requests of the form:

```
=3n"C Stack Backtrace"
```

This spaces three lines and displays the literal string. The request `$v` displays all non-zero `adb` variables. The request `0$s` sets the maximum offset for symbol matches to zero, thus suppressing the display of symbolic labels in favor of octal values. Note that this is only done for displaying the data segment. The request

```
<b,-1/8ona
```

displays a dump from the base of the data segment to the end-of-file with an octal address field and 8 octal numbers per line.

Accounting File Dump

As another illustration, consider a set of requests to dump the contents `/etc/utmp` or `/usr/adm/wtmp`, both of which are composed of 8-character terminal names, 8-character login names, 16-character host names, and a 4-byte integer representing the login time.

```
% adb /etc/utmp -
0,-1?cccccccc8tcccccccc8tcccccccccccccccc16tYn
```

The `c` format is repeated 8 times, 8 times, and 16 times. The `8t` means go to align on an 8-character-position boundary, and `16t` means to align on a 16-character-position boundary. `Y` causes the 4-byte integer representing the login time to print in `ctime`(3) format.

Converting Values

You can use `adb` to convert values from one representation to another. For example, to print the hexadecimal number `ff` in octal, decimal, and hexadecimal, type:

```
ff = odx
377  255  #ff
```

The default input radix of `adb` is hexadecimal. Formats are remembered, so that typing subsequent numbers will display them in the same format. Character values may be converted as well:

```
'a' = oc
    0141    a
```

This technique may also be used to evaluate expressions, but be warned that all binary operators have the same precedence, which is lower than for unary operators.

6.5. Patching

Patching files with `adb` is accomplished with the write requests `w` or `W`. This is often used in conjunction with the locate requests `l` or `L`. In general, the syntax for these requests is as follows:

```
?l value
```

The `l` matches on two bytes, whereas `L` matches four bytes. The `w` request writes two bytes, whereas `W` writes four bytes. The value field in either locate or write requests is an expression. Either decimal and octal numbers, or character strings, are permitted.

In order to modify a file, `adb` must be invoked as follows:

```
% adb -w file1 file2
```

When invoked with this option, *file1* and *file2* are created if necessary, and opened for both reading and writing.

NOTE *The* `$W` *command has the same effect during an* `adb` *session as the* `-w` *option used on the command line.*

For example, consider the following C program, `zen.c`: We will change the word "Thys" to "This" by compiling `zen`.

```
char    str1[]  = "Thys 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;
}
```

Use the following requests:

```
% adb -w zen -
<b?l 'Th'
?W 'This'
```

The request `<b?l` starts at the start of the data segment 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 object file. The form **?*** would be used for a 410-format file.

More frequently the request is typed as:

```
?l 'Th'; ?s
```

which locates the first occurrence of "Th", and display the entire string. Execution of this `adb` request sets dot to the address of those characters in the string.

NOTE *When using the* `?l` *or* `?L` *commands, be cautious of gaps in the address range that you want to search.*

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 using adb, before running the program. For example:

```
% adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single-step mode. In this case it starts 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.

6.6. Anomalies

Below is a list of some strange things that users should be aware of.

1) When displaying addresses, adb uses either text or data symbols from the object file. This sometimes causes unexpected symbol names to be displayed with data (for example, savr5+022). This does not happen if **?** is used for text (instructions) and **/** for data.

2) The adb debugger cannot handle C register variables in the most recently activated function.

7

Sun386*i* adb Tutorial

7.1. A Quick Survey

Available on most UNIX systems, adb is a debugger that permits you to examine core files resulting from aborted programs, display 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. The reader is expected to be familiar with basic SunOS commands, and with the C language.

Starting adb

Start adb with a shell command like

```
% adb objectfile corefile
```

where *objectfile* is a SunOS executable file and *corefile* is a core dump file. If you leave object files in a.out, then the invocation is simple:

```
% adb
```

If you place object files into a named *program*, then the invocation is a bit harder:

```
% adb program
```

The filename minus (–) means ignore the object file argument, as in:

```
% adb - core
```

This is for examining the core file without reference to an object file. adb provides requests for examining locations in either file: **?** examines the contents of *objectfile*, while **/** examines the contents of *corefile*. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

Current Address

adb maintains a current address, called **dot**. When an address is entered, the current address is set to that location, so that

```
0126?i
```

sets dot to octal 126 and displays the instruction at that address. The request

```
.,10/d
```

displays 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item displayed. When used with the **?** or **/** requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up of decimal integers, octal integers, hexadecimal integers, and symbols from the program under test. These may be combined with the operators **+** (plus), **-** (minus), ***** (multiply), **%** (integer divide), **&** (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, you can type name.

Formats

To display data, specify a sequence of letters and characters to describe the format of the display. Formats are remembered, in the sense that typing a request without a format displays the new output in the previous format. Here are the most commonly used format letters:

Table 7-1 *Some* adb *Format Letters*

Letter	***Description***
b	one byte in octal
B	one byte in hex
c	one byte as a character
o	one word in octal
d	one word in decimal
f	one single-precision floating point value
i	Sun386*i* instruction
s	a null-terminated character string
a	the value of dot
u	one word as an unsigned integer
n	print a newline
r	print a blank space
^	backup dot (not really a format)
+	advance dot (not really a format)

Format letters are also available for long values: for example, D for long decimal, and F for double-precision floating point. Since integers are long-words on the Sun, capital letters are used more often than not.

General Request Meanings

The general form of a request is:

address , *count command modifier*

which sets dot to *address* and executes *command count* times. The following table illustrates some general adb command meanings:

Table 7-2 *Some* adb *Commands*

Command	Meaning
?	Print contents from object file
/	Print contents from core file
=	Print value of expression
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

Since adb catches signals, you cannot use a quit signal to exit from adb. The request $q or $Q (or [CTRL-D]) must be used to exit from adb.

7.2. Debugging C Programs on Sun386*i*

If you use adb because you are accustomed to it, you will want to compile programs with the -go option, to produce old-style symbol tables. This will make debugging proceed according to expectations.

Debugging A Core Image

Consider the C program below, which illustrates a common error made by C programmers. The object of the program is to change the lower case t to an upper case T in the string pointed to by ch, and then write the character string to the file indicated by the first argument.

```
#include <stdio.h>

char *cp = "this is a sentence.";

main(argc, argv)
int argc;
char **argv;
{
    FILE *fp;
    char c;

    if (argc == 1) {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }
    if ((fp = fopen(argv[1], "w")) == NULL) {
        perror(argv[1]);
        exit(2);
    }
    cp = 'T';
```

```
    while (c = *cp++)
        putc(c, fp);
    fclose(fp);
    exit(0);
}
```

The bug is that the character `T` is stored in the pointer `cp` instead of in the string pointed to by `cp`. Compile the program as follows:

```
% cc -go example1.c
% a.out junk
Segmentation fault (core dumped)
```

Executing the program produces a core dump because of an out-of-bounds memory reference. Now invoke `adb` by typing:

```
% adb
core file = core -- program "a.out"
memory fault
```

Commonly the first debugging request given is

```
$c
main[8074](2,fffd7c,fffd88) + 92
```

which produces a C backtrace through the subroutines called. The output from `adb` tells us that only one function — `main` — was called, and the arguments `argc` and `argv` have the hexadecimal values 2 and `fffd7c` respectively. Both these values look reasonable — 2 indicates two arguments, and `fffd7c` equals the stack address of the parameter vector. The next request

```
$C
main[8074](2,fffd7c,fffd88) + 92
    fp:         10468
    c:          104
```

generates a C backtrace plus an interpretation of all the local variables in each function, and their values in hexadecimal. The value of the variable `c` looks incorrect since it is outside the ASCII range. The request

```
$r
gs      0xfbff0000                      ecx     0x28680
fs      0xfbff0000                      eax     0x54
es      0xfcff0083                      retaddr 0xfc06e38e
ds      0x83                            trapno  0xe
edi     0x30890                         err     0x4
esi     0x28680                         eip     0x120b          main+0x10f
ebp     0xfbfffec8                      cs      0x7b
esp     0xfcff97e0                      efl     0x10206         end+0x7202
ebx     0x2a0c0                         uesp    0xfbfffec0
edx     0xfbfffe6a                      ss      0x83
main+0x10f:     movb    (%eax),%al
```

displays the registers, including the program counter, and an interpretation of the instruction at that location. The request

```
$e
cp:             0x55
_exit_nhandlers:            0x0
_exit_tnames:               0x35dc
_ctype_:        0x20202000
_smbuf:         0x65c0
_iob:           0x0
__mallinfo:     0x0
_root:          0x0
_lbound:        0x0
_ubound:        0x0
curbrk:         0x9004
errno:          0x0
environ:        0xfbfffef4
end:            0x0
```

displays the values of all external variables.

A map exists for each file handled by adb. The map for a.out files is referenced by **?** whereas the map for core files is referenced by **/**. Furthermore, a good rule of thumb is to use **?** for instructions and **/** for data when looking at programs. To display information about maps, type:

```
$m
b1 = 8000              e1 = b000              f1 = 800
b2 = 10000             e2 = 11000             f2 = 3800
/ map        'core'
b1 = 10000             e1 = 13000             f1 = 1800
b2 = fff000            e2 = 1000000           f2 = 4800
```

This produces a report of the contents of the maps. More about these maps later.

In our example, we might want to see the contents of the string pointed to by cp. We would want to see the string pointed to by cp in the core file:

```
*cp/s
55:
data address not found
```

Because the pointer was set to 'T' (hex 54) and then incremented, it now equals hex 55. On the Sun386*i*, there is nothing mapped at this address, so the data at address 55 cannot be found. We could also display information about the arguments to a function. To get the decimal value of the argc argument to main, which is a long integer, type:

```
main.argc/D
fffd6c:         2
```

To display the hex values of the three consecutive cells pointed to by argv in the function main, type:

```
*main.argv,3/X
fffd7c:         fffdc0          fffdc6          0
```

Note that these values are the addresses of the arguments to main. Therefore, typing these hex values should yield the command-line arguments:

```
fffdc0/s
fffdc0:         a.out
```

The request:

```
.=
                fffdc0
```

displays the current address (not its contents) in hex, which has been set to the address of the first argument. The current address, dot, is used by adb to remember its current location. It allows the user to reference locations relative to the current address. For example

```
 .+6/s
fffdc6:         zzz
```

prints the first command-line argument.

Setting Breakpoints

You set breakpoints in a program with the :b instruction, which has this form:

```
address:b [request]
```

Consider the C program below, which changes tabs into blanks, and is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

```
#include <stdio.h>

#define MAXLIN  80
#define YES 1
#define NO  0
#define TABSP   8

int tabs[MAXLIN];

main()
{
    int *ptab, col, c;

    ptab = tabs;
    settab(ptab);   /* set initial tab stops */
    col = 1;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '\t':
            while (tabpos(col) != YES) {
                putchar(' ');
                col++;
            }
            putchar(' ');
            col++;
            break;
        case '\n':
            putchar('\n');
            col = 1;
            break;
        default:
            putchar(c);
            col++;
        }
    }
    exit(0);
}

tabpos(col) /* return YES if col is a tab stop, NO if not */
int col;
{
    if (col > MAXLIN)
        return(YES);
    else
        return(tabs[col]);
}

settab(tabp)    /* set initial tab stops every TABSP spaces */
int *tabp;
{
    int i;

    for (i = 0; i <= MAXLIN; i++)
        (i % TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Run the program under the control of adb, and then set two breakpoints as follows:

```
% adb a.out -
settab+5:b
tabpos+5:b
```

This sets breakpoints at the start of the two functions. Sun compilers generate statement labels only with the –g option, which is incompatible with adb. In adb, you can set breakpoints anywhere, but you can only refer to a breakpoint as a function entry point plus an offset. To display the location of breakpoints, type:

```
$b
breakpoints
count   bkpt            command
1       tabpos+5
1       settab+5
```

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 this example no command fields are present.

Display the instructions at the beginning of function settab() in order to observe that the breakpoint is set after the link assembly instruction:

```
settab,5?ia
settab:
settab:         jmp     settab+0x58
settab+5:       movl    $0,-4(%ebp)
settab+0xc:     jmp     settab+0x48
settab+0x11:    movl    -4(%ebp),%eax
settab+0x14:    movl    $8,%ecx
settab+0x19:
```

This request displays five instructions starting at settab with the address of each location displayed. Another variation is

```
settab,5?i
settab:
settab:                 jmp     settab+0x58
                movl    $0,-4(%ebp)
                jmp     settab+0x48
                movl    -4(%ebp),%eax
                movl    $8,%ecx
```

which displays the instructions with only the starting address. Note that we accessed the addresses from a.out with the ? command. In general, when asking for a display of multiple items, adb advances the current address the number of bytes necessary to satisfy the request; in the above example, five instructions were displayed and t he current address was advanced 26 bytes.

To run the program, type:

```
:r
```

To delete a breakpoint, for instance the entry to the function `tabpos()`, type:

```
tabpos:d
```

Once the program has stopped, in this case at the breakpoint for `settab()`, `adb` requests can be used to display the contents of memory. To display a stack trace, for example, type:

```
$c
settab[8250](10658) + 4
main[8074](1,fffd84,fffd8c) + 1a
```

And to display three lines of eight locations each from the array called `tabs`, type:

```
tabs,3/8X
tabs:
tabs:       0       0       0       0       0       0       0       0
            0       0       0       0       0       0       0       0
            0       0       0       0       0       0       0       0
```

At this time (at location `settab`) the `tabs` array has not yet been initialized. If you just deleted the breakpoint at `tabpos`, put it back by typing:

```
tabpos:b
```

To continue execution of the program from the breakpoint type:

```
:c
    x
```

You will need to give the `a.out` program a line of data, as in the figure above. Once you do, it will encounter a breakpoint at `tabpos+4` and stop again. Examine the `tabs` array once more: now it is initialized, and has a one set in every eighth location:

```
tabs,3/8X
tabs:
tabs:       1       0       0       0       0       0       0       0
            1       0       0       0       0       0       0       0
            1       0       0       0       0       0       0       0
```

You will have to type `:c` eight more times in order to get your line of output, since there is a breakpoint at every input character. Type [CTRL-D] to terminate the `a.out` process; you are back in command-level of `adb`.

Advanced Breakpoint Usage

The 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 passed on to the test program if you type:

```
:c 0
```

Now let's reset the breakpoint at `settab()` and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+5:b settab,5?ia
:r
settab,5?ia
settab:
settab:           jmp    settab+0x58
settab+5:         movl   $0,-4(%ebp)
settab+0xc:       jmp    settab+0x48
settab+0x11:      movl   -4(%ebp),%eax
settab+0x14:      movl   $8,%ecx
settab+0x19:
breakpoint        settab+5:         movl   $0,-4(%ebp)
```

It is possible to stop every two breakpoints, if you type `,2` before the breakpoint command. Variables can also be displayed at the breakpoint, as illustrated below:

```
tabpos+4,2:b main.col?X
:c
    x
fffd64:           1
fffd64:           2
breakpoint        tabpos+5:         movl    $0x50,%eax
```

This shows that the local variable `col` changes from 1 to 2 before the occurrence of the breakpoint.

NOTE *Setting a breakpoint causes the value of dot to be changed. However, executing the program under* `adb` *does not change the value of dot.*

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b main.ptab/X; main.c/X
:r
fffd68:           10658
fffd60:           0
breakpoint        settab+5:         movl   $0,-4(%ebp)
```

The semicolon is used to separate multiple `adb` requests on a single line.

Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as follows. This request kills any existing program under test and starts `a.out` afresh:

```
:r arg1 arg2 ... <infile >outfile
```

The program being debugged can be single stepped as follows. If necessary, this request starts up the program being debugged and stops after executing the first instruction:

```
:s
```

You can enter a program at a specific address by typing:

```
address:r
```

The count field can be used to skip the first *n* breakpoints, as follows:

```
,n:r
```

This request may also be used for skipping the first *n* breakpoints when continuing a program:

```
,n:c
```

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
```

7.3. File Maps

SunOS supports several executable file formats.

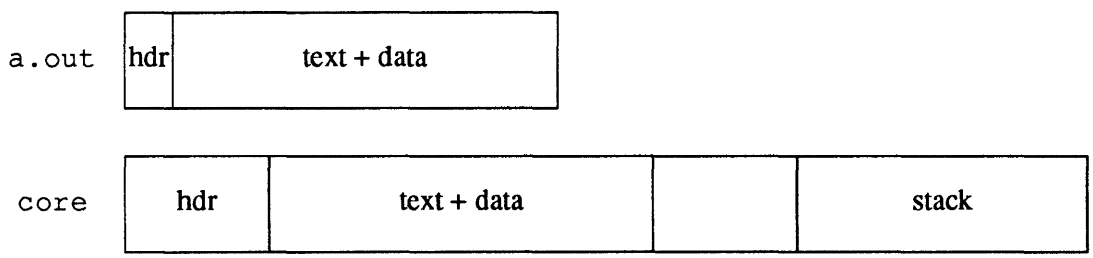
NOTE *On the Sun386i, all executable files are COFF files. An additional COFF header precedes the a.out header; this* `a.out` *header is slightly different than the Sun-3 or Sun-4* `a.out` *header. However, the executable file types are identical.*

Executable type 407 is generated by the `cc` (or `ld`) flag `-N`. Executable type 410 is generated by the flag `-n`. An executable type 413 is generated by the flag `-z`; the default is type 413. `adb` interprets these different file formats, and provides access to the different segments through a set of maps. To display the maps, type **$m** from inside `adb`.

407 Executable Files

In 407-format files, instructions and data are intermixed. This makes it impossible for `adb` to differentiate data from instructions, but `adb` will happily display in either format. Furthermore, some displayed symbolic addresses look incorrect (for example, data addresses as offsets from routines). Here is a picture of 407-format files:

Figure 7-1 *Executable File Type 407*



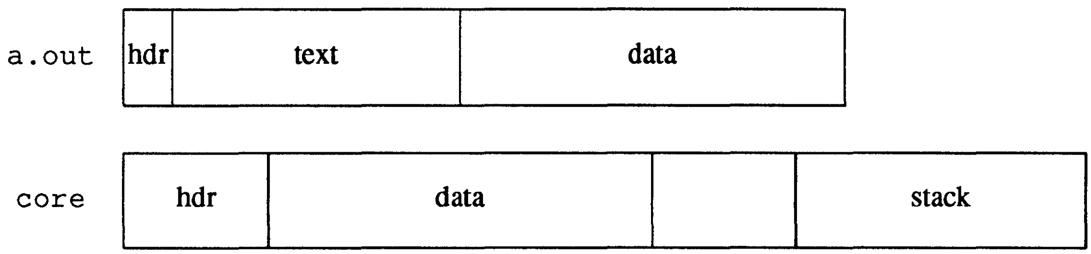
Here are the maps and variables for 407-format files:

```
$m
? map          'a.out'
b1 = 8000                e1 = 8f28                 f1 = 20
b2 = 8000                e2 = 9560                 f2 = 20
/ map          'core'
b1 = 8000                e1 = b800                 f1 = 1800
b2 = fff000              e2 = 1000000              f2 = 5000
$v
variables
b = 0100000
d = 03070
e = 0407
m = 0407
s = 010000
t = 07450
```

410 Executable Files

In 410-format files (pure executable), instructions are separate from data. The ? command accesses the data part of the a.out file, telling adb to use the second part of the map in that 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. Here is a picture of 410-format files:

Figure 7-2 *Executable File Type 410*



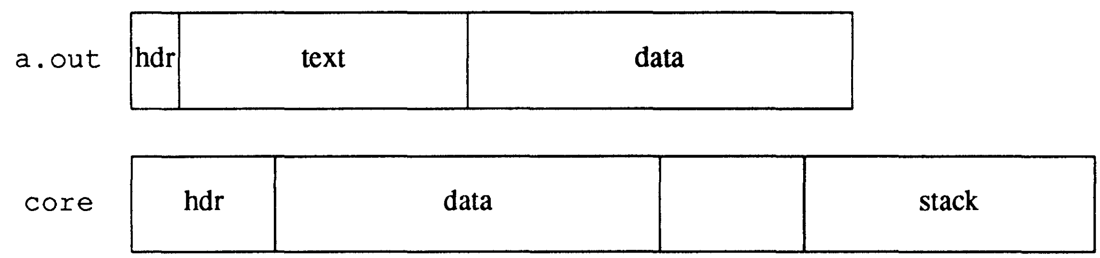
Here are the maps and variables for 410-format files:

```
$m
? map        'a.out'
b1 = 8000            e1 = 8f28             f1 = 20
b2 = 10000           e2 = 10638            f2 = f48
/ map        'core'
b1 = 10000           e1 = 12800            f1 = 1800
b2 = fff000          e2 = 1000000          f2 = 4000
$v
variables
b = 0200000
d = 03070
e = 0410
m = 0410
s = 010000
t = 07450
```

413 Executable Files

In 413-format files (pure demand-paged executable) the instructions and data are also separate. However, in this case, since data is contained in separate pages, the base of the data segment is also relative to address zero. In this case, since the addresses overlap, it is necessary to use the `?*` operator to access the data space of the `a.out` file. In both 410 and 413-format files the corresponding core file does not contain the program text. Here is a picture of 413-format files:

Figure 7-3 *Executable File Type 413*



The only difference between a 410 and a 413-format file is that 413 segments are rounded up to page boundaries. Here are the maps and variables for 413-format files:

```
$m
? map         'abort'
b1 = 8000              e1 = 9000               f1 = 800
b2 = 10000             e2 = 10800              f2 = 1800
/ map         'core'
b1 = 10000             e1 = 12800              f1 = 1800
b2 = fff000            e2 = 1000000            f2 = 4000
$v
variables
b = 0200000
d = 04000
e = 0413
m = 0413
s = 010000
t = 010000
```

Variables

The b, e, and f fields are used to map addresses into file addresses. The f1 field is the length of the header at the beginning of the file — 020 bytes for an a.out file and 02000 bytes for a core file. The f2 field is the displacement from the beginning of the file to the data. For a 407-format file with mixed text and data, this is the same as the length of the header; for 410 and 413-format files, this is the length of the header plus the size of the text portion. The b and e fields are the starting and ending locations for a segment. Given the address *A*, the location in the file (either a.out or core) is calculated as:

```
b1<A<e1    file address = (A-b1)+f1
b2<A<e2    file address = (A-b2)+f2
```

You can access locations by using the adb-defined variables. The $v request displays the variables initialized by adb:

- b base address of data segment,
- d length of the data segment,
- s length of the stack,
- t length of the text,
- m execution type (407, 410, 413).

Those variables not presented are zero. Use can be made of these variables by expressions such as

```
<b
```

in the address field. Similarly, the value of a variable can be changed by an assignment request such as

```
02000>b
```

which sets b to octal 2000. These variables are useful to know if the file under examination is an executable or core image file.

The adb program 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.

7.4. Advanced Usage

One of the uses of adb is to examine object files without symbol tables; dbx cannot handle this kind of task. With adb, you can even combine formatting requests to provide elaborate displays. Several examples are given below.

Formatted Dump

The following adb command line displays four octal words followed by their ASCII interpretation from the data space of the core file:

```
<b,-1/4o4^8Cn
```

Broken down, the various requests 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^ — Back up 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 displayed as followed by the corresponding character in the range 0140 to 0177. An @ is displayed as @@.

n — Print a newline.

The following request could have been used instead to allow the displaying to stop at the end of the data segment.

```
<b,<d/404^8Cn
```

The request <d provides the data segment size in bytes. Because adb can read in scripts, you can use formatting requests to produce image dump scripts. Invoked adb as follows:

```
% adb a.out core < dump
```

This reads in a script file, dump, containing formatting requests. Here is an example of such a script:

```
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,-1/8ona
```

The request `120$w` sets the width of the output to 120 characters (normally, the width is 80 characters). `adb` attempts to display addresses as:

```
symbol + offset
```

The request `4095$s` increases the maximum permissible offset to the nearest symbolic address from the default 255 to 4095. The request = can be used to display literal strings. Thus, headings are provided in this dump program with requests of the form:

```
=3n"C Stack Backtrace"
```

This spaces three lines and displays the literal string. The request `$v` displays all non-zero `adb` variables. The request `0$s` sets the maximum offset for symbol matches to zero, thus suppressing the display of symbolic labels in favor of octal values. Note that this is only done for displaying the data segment. The request

```
<b,-1/8ona
```

displays a dump from the base of the data segment to the end-of-file with an octal address field and 8 octal numbers per line.

Accounting File Dump

As another illustration, consider a set of requests to dump the contents `/etc/utmp` or `/usr/adm/wtmp`, both of which are composed of 8-character terminal names, 8-character login names, 16-character host names, and a 4-byte integer representing the login time.

```
% adb /etc/utmp -
0,-1?cccccccc8tcccccccc8tcccccccccccccccc16tYn
```

The `c` format is repeated 8 times, 8 times, and 16 times. The `8t` means go to the 8th tab stop, and `16t` means to the 16th tab stop. `Y` causes the 4-byte integer representing the login time to print in `ctime`(3) format.

Converting Values

You can use `adb` to convert values from one representation to another. For example, to print the hexadecimal number `ff` in octal, decimal, and hexadecimal, type:

```
ff = odx
    072 58  #3a
```

The default input radix of `adb` is hexadecimal. Formats are remembered, so that typing subsequent numbers will display them in the same format. Character values may be converted as well:

```
'a' = oc
    0141    a
```

This technique may also be used to evaluate expressions, but be warned that all binary operators have the same precedence, which is lower than for unary operators.

7.5. Patching

Patching files with adb is accomplished with the write requests w or W. This is often used in conjunction with the locate requests l or L. In general, the syntax for these requests is as follows:

```
?l value
```

The l matches on two bytes, whereas L matches four bytes. The w request writes two bytes, whereas W writes four bytes. The value field in either locate or write requests is an expression. Either decimal and octal numbers, or character strings, are permitted.

In order to modify a file, adb must be invoked as follows:

```
% adb -w file1 file2
```

When invoked with this option, *file1* and *file2* are created if necessary, and opened for both reading and writing.

For example, consider the following C program, zen.c: We will change the word "Thys" to "This" in the executable file.

```
char    str1[]  = "Thys 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;
}
```

Use the following requests:

```
% adb -w zen -
?l 'Th'
?W 'This'
```

The request ?l starts a 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 be used for a 411 file.

More frequently the request is typed as

```
?l 'Th'; ?s
```

which locates the first occurrence of "Th", and display the entire string. Execution of this adb request sets dot to the address of those characters in the string.

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 using adb, before running the program. For example:

```
% adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single step mode. In this case it starts 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 caused flag to be changed in the memory of the subprocess.

7.6. Anomalies

Below is a list of some strange things that users should be aware of.

1) When displaying addresses, adb uses either text or data symbols from the a.out file. This sometimes causes unexpected symbol names to be displayed with data (for example, savr5+022). This does not happen if **?** is used for text (instructions) and **/** for data.

2) The adb debugger cannot handle C register variables in the most recently activated function.

8

adb Reference

adb [**-w**] [**-k**] [**-I** *dir*] [*objectfile* [*corefile*]]

adb is an interactive, general-purpose, assembly-level debugger, that examines files and provides a controlled environment for executing SunOS programs.

Normally *objectfile* is an executable program file, preferably containing a symbol table generated by the compiler's -go option. If the file does not contain a symbol table, it can still be examined, but the symbolic features of adb cannot be used. The default *objectfile* is a.out.

The *corefile* is assumed to be a core image file produced by executing *objectfile* and having a problem causing the core image to be dumped to the file *core*. The default *corefile* is core.

8.1. adb Options

-w If either *objectfile* or *corefile* does not exist, create the nonexistent file and open both files for reading and writing.

-k Do SunOS kernel memory mapping; should be used when *corefile* is a SunOS crash dump or /dev/mem.

-I Specifies a directory where files to be read with $< or $<< (see below) will be sought; the default is /usr/lib/adb.

8.2. Using adb

adb reads commands from the standard input and displays responses on the standard output, ignoring QUIT signals. An INTERRUPT signal returns to the next adb command.

adb saves and restores terminal characteristics when running a subprocess. This makes it possible to debug programs that manipulate the screen. See *tty*(4).

In general, requests to adb are of the form

[*address*] [, *count*] [*command*] [;]

The symbol **dot** (.) represents the current location. It is initially zero. If *address* is present, then **dot** is set to *address*. For most commands *count* specifies how many times the command is to be executed. The default *count* is 1 (one). Both *address* and *count* may be expressions.

8.3. adb Expressions

. The value of **dot**.

+ The value of **dot** incremented by the current increment.

^ The value of **dot** decremented by the current increment.

& The last *address* typed; this used to be ".

integer
A number. The prefixes 0o and 0O (zero oh) force interpretation in octal radix; the prefixes 0t and 0T force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 0o20= 0t16= 0x10= sixteen. If no prefix appears, then the *default radix* is used; see the $d command. The default radix is initially hexadecimal. Hexadecimal digits are 0123456789abcdefABCDEF with the obvious values. Note that if a hexadecimal number starts with a letter, but does not duplicate a defined symbol, it is accepted as a hexadecimal value. To enter a hexadecimal number that is the same as a defined symbol, precede it by 0, 0x, or 0X.

'cccc'
The ASCII value of up to 4 characters. A backslash (\) may be used to escape a '.

<name
The value of *name*, which is either a variable name or a register name; adb maintains several variables (see VARIABLES) named by single letters or digits. If *name* is a register name, then the value of the register is obtained from the system header in *corefile*. The register names are those printed by the $r command.

symbol
A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The backslash character (\) may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objectfile*. An initial _ will be prepended to *symbol* if needed.

_symbol
In C, the true name of an external symbol begins with underscore (_). It may be necessary to use this name to distinguish it from internal or hidden variables of a program.

NOTE *_symbol applies only to Sun-3 and Sun-4 systems. It is not used on Sun386i systems.*

routine.name
The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*. Works only if the program has been compiled using the -go flag. See *cc*(1).

e *s* Sun386*i* only. Like *s*, but steps over subroutine calls instead of into them.

(*expr*) The value of the expression *expr*.

Unary Operators

* *expression*
The contents of the location addressed by *expression* in *corefile.*

% *expression*
The contents of the location addressed by *expression* in *objectfile* (used to be @).

–*expression*
Integer negation.

~ *expression*
Bitwise complement.

expression
Logical negation.

^F *expression*
(Control–f) Translates program addresses into source file addresses. Works only if the program has been compiled using the `-go` flag. See *cc*(1).

^A*expression*
(Control–a) Translates source file addresses into program addresses. Works only if the program has been compiled using the `-go` flag. See *cc*(1).

\`*name*
(Back–quote) Translates a procedure name into a source file address. Works only if the program has been compiled using the `-go` flag. See *cc*(1).

"*filename*"
A filename enclosed in quotation marks (for instance, `main.c`) produces the source file address for the zero-th line of that file. Thus to reference the third line of the file main.c, we say: `"main.c"+3`. Works only if the program has been compiled using the `-go` flag. See *cc*(1).

Binary Operators

Binary operators are left associative and are less binding than unary operators.

expression-1 + *expression-2*
Integer addition.

expression-1 – *expression-2*
Integer subtraction.

expression-1 * *expression-2*
Integer multiplication.

expression-1 % *expression-2*
Integer division.

expression-1 & *expression-2*
Bitwise conjunction.

expression-1 | *expression-2*
Bitwise disjunction.

expression-1 # *expression-2*
Expression1 rounded up to the next multiple of *expression2*.

8.4. adb Variables

adb provides several variables. Named variables are set initially by adb but are not used subsequently. Numbered variables are reserved for communication as follows:

0 The last value printed.

1 The last offset part of an instruction source.

2 The previous value of variable 1.

9 The count on the last $< or $<< command.

On entry the following are set from the system header in the *corefile*. If *corefile* does not appear to be a core file then these values are set from *objectfile*.

b The base address of the data segment.

d The data segment size.

e The entry point.

m The magic number (0407, 0410 or 0413), depending on the file's type.

s The stack segment size.

t The text segment size.

8.5. adb Commands

Commands to adb consist of a *verb* followed by a *modifier* or list of modifiers.

adb Verbs

The verbs are:

? Print locations starting at *address* in *objectfile*.

/ Print locations starting at *address* in *corefile*.

= Print the value of *address* itself.

@ Interpret *address* as a source file address, and print locations in *objectfile* or lines of the source text. Works only if the program has been compiled using the -go flag. See cc(1).

: Manage a subprocess.

$ Execute miscellaneous commands.

> Assign a value to a variable or register.

RETURN
Repeat the previous command with a *count* of 1. *Dot* is incremented by its current increment.

! Call the shell to execute the following command.

Each verb has a specific set of **modifiers** ; these are described below.

?, /, @, and = Modifiers

The first four verbs described above take the same *modifiers*, which specify the format of command output. Each modifier consists of a format letter (*fletter*) preceded by an optional repeat count (*rcount*). Each verb can take zero, one, or more modifiers.

{ ?, /, @, = } [[*rcount*] *fletter* ...]

Each modifier specifies a format that increments *dot* by a certain amount, which is given below. If a command is given without a modifier, the last specified format is used to display output. The following table shows the format letters, the amount they increment *dot*, and a description of what each letter does. Note that all octal numbers output by adb are preceded by 0.

Format	*Dot+=*	*Description*
o	2	Print 2 bytes in octal.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long in signed octal.
d	2	Print in decimal.
D	4	Print long in decimal.
x	2	Print 2 bytes in hexadecimal.
X	4	Print 4 bytes in hexadecimal.
h	2	Sun386*i* only. Print 2 bytes in hexadecimal in reverse order.
H	4	Sun386*i* only. Print 4 bytes in hexadecimal in reverse order.
u	2	Print as an unsigned decimal number.
U	4	Print long as an unsigned decimal.
f	4	Print the 32-bit value as a floating point number.
F	8	Print the 64-bit number as a double floating point number.
b	1	Print the addressed byte in octal.
B	1	Sun386*i* only. Print the addressed byte in hexadecimal.
c	1	Print the addressed character.
C	1	Print the addressed character using the standard escape convention. Print control characters as ^X and the delete character as ^?.
s	*n*	Print the addressed characters until a null character is reached; *n* is the length of the string including its zero terminator.

Format	*Dot+=*	*Description*
`S`	*n*	Print string using the escape conventions of `C`; *n* is the length of the string including its zero terminator.
`Y`	4	Print 4 bytes in *ctime*(3) format.
`i`	*n*	Print as machine instructions; *n* is the number of bytes occupied by the instruction. In this format, variables 1 and 2 are set to the offset parts of the source and destination, respectively.
`M`	*n*	Sun386*i* only. Print as machine instructions along with machine code; *n* is the number of bytes occupied by the instruction. In this format, variables 1 and 2 are set to the offset parts of the source and destination, respectively.
`z`	*n*	Print as machine instructions with MC68010 Sun-2 instruction timings; *n* is the number of bytes occupied by the instruction. In this format, variables 1 and 2 are set to the offset parts of the source and destination respectively.
`I`	0	Print the source text line specified by *dot* (@ command), or most closely corresponding to *dot* (? command).
`a`	0	Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below: / local or global data symbol ? local or global text symbol = local or global absolute symbol
`p`	4	Print the addressed value in symbolic form using the same rules for symbol lookup as for `a`.
`A`	0	Print the value of *dot* in source file-symbolic form, that is: `"file"+nnn`. Works only if the program has been compiled with the `-go` flag. See *cc*(1).
`P`	4	Print the addressed value in source-file symbolic form, that is: `"file"+nnn`. Works only if the program has been compiled using the `-go` flag. See *cc*(1).
`t`	0	When preceded by an integer, tabs to the next appropriate tab stop. For example, `8t` moves to the next 8-space tab stop.
`r`	0	Print a space.
`n`	0	Print a newline.
`"..."`	0	Print the enclosed string.
`^`	0	*Dot* decremented by current increment; nothing is printed.
`+`	0	*Dot* incremented by 1; nothing is printed.
`-`	0	*Dot* decremented by 1; nothing is printed.

? and / Modifiers

Only the verbs ? and / take the following modifiers:

[?/]l *value mask*
Words starting at *dot* are masked with *mask* and compared to *value* until a match is found. If the command is L instead of l, the match is for 4 bytes at a time instead of 2. If no match is found *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted then –1 is used.

[?/]w *value* ...
Write the 2-byte *value* into the addressed location. If the command is W instead of w, write 4 bytes instead of 2. If the command is v, write only 1 byte. Odd addresses are not allowed when writing to the subprocess address space.

[?/]m *b1 e1 f1* [?/]
New values for (*b1, e1, f1*) are recorded. If fewer than three expressions are given, then the remaining map parameters are left unchanged. If the ? or / is followed by *, then the second segment (*b2, e2, f2*) of the address mapping is changed (see *Address Mapping* below). If the list is terminated by ? or /, then the file, *objectfile* or *corefile* respectively, is used for subsequent requests. For example, /m? causes / to refer to *objectfile*.

: Modifiers

Only the verb : takes the following modifiers:

a *cmd* — Sun386*i* only. Set a data access breakpoint at *address*. Like *b* except that the breakpoint is hit when the program reads or writes to *address*.

b *cmd* — Set breakpoint at *address*. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is encountered the command *cmd* is executed. If this command is omitted or sets *dot* to zero, then the breakpoint causes a stop.

w — Sun386*i* only. Set a data write breakpoint at *address*. Like *b* except that the breakpoint is hit when the program writes to *address*.

B *c* — Like *b* but takes a source file address. Works only if the program has been compiled using the –go flag. See *cc*(1).

d — Delete breakpoint at *address*.

D — Like *d* but takes a source file address. Works only if the program has been compiled using the –go flag. See *cc*(1).

z — Sun386*i* only. Delete all breakpoints.

r — Run *objectfile* as a subprocess. If *address* is given explicitly, then the program is entered at this point; otherwise, the program is entered at its standard entry point. An optional *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are enabled on entry to the subprocess.

`c` *s* The subprocess is continued with signal *s*; see `sigvec`(2). If *address* is given then the subprocess is continued at this address. If no signal is specified, then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for `r`.

`s` *s* Same as for `c` except that the subprocess is single stepped *count* times. If there is no current subprocess, then *objectfile* is run as a subprocess as for `r`. In this case no signal can be sent; the remainder of the line is treated as an argument list for the subprocess.

`S` Like s but single steps by source lines, rather than by machine instructions. This is achieved by repeatedly single-stepping machine instructions until the corresponding source file address changes. Thus procedure calls cause stepping to stop. Works only if the program has been compiled using the `-go` flag. See `cc`(1).

`u` Sun386*i* only. Continue uplevel, stopping after the current routine has returned. Should only be given after the frame pointer for the current routine has been pushed on the stack.

`i` Add the signal specified by *address* to the list of signals that are passed directly to the subprocess with the minimum of interference. Normally, `adb` intercepts all signals destined for the subprocess, and the `:c` command must be issued to continue the process with the signal. Signals on this list are handed to the process with an implicit `:c` commands as soon as they are seen.

`t` Remove the signal specified by *address* from the list of signals that are implicitly passed to the subprocess.

`k` Terminate (kill) the current subprocess, if any.

`A` Sun386*i* only. Attach the process whose process ID is given by *address*. The PID is generally preceded by `0t` so that it will be interpreted in decimal.

`R` Sun386*i* only. Release (detach) the current process.

`$` Modifiers

Only the verb `$` takes the following modifiers:

<*file* Read commands from *file*. If this command is executed in a file, further commands in the file are not seen. If *file* is omitted, the current input stream is terminated. If a *count* is given, and it is zero, the

command is ignored. The value of the count is placed in variable *9* before the first command in *file* is executed.

<< *file* — Similar to <, but can be used in a file of commands without closing the file. Variable *9* is saved during the execution of this command, and restored when it completes. Not more than 5 << files that can be open simultaneously.

> *file* — Append output to *file*, which is created if it does not exist. If *file* is omitted, output is returned to the terminal.

`?` — Print the process id, the signal that stopped the subprocess, and the registers. Produces the same response as `$` used without any modifier.

`r` — Print the general registers and the instruction addressed by the program counter; *dot* is set to that address.

`b` — Print all breakpoints and their associated counts and commands.

`c` — C stack backtrace. If *address* is given, it is taken as the address of the current frame instead of the contents of the frame-pointer register. If *count* is given, only the first *count* frames are printed.

`C` — Similar to `c`, but in addition prints the names and 32-bit values of all automatic and static variables for each active function. Works only if the program has been compiled using the `-go` flag. See `cc`(1).

`d` — Set the default radix to *address* and report the new value. Note that *address* is interpreted in the (old) current radix. Thus `10$d` never changes the default radix. To make the default radix decimal, use `0t10$d`.

`e` — Print the names and values of external variables.

`w` — Set the page width for output to *address* (default 80).

`s` — Set the limit for symbol matches to *address* (default 255).

`o` — Regard all input integers as octal.

`q` — Exit `adb`.

`v` — Print all non-zero variables in octal.

`m` — Print the address map.

`f` — Print a list of known source file names.

`p` — Print a list of known procedure names.

`p` — For kernel debugging. Change the current kernel memory mapping to map the designated *user structure* to the address given by the symbol `_u`. The *address* argument is the address of the user's `proc` structure.

`i` — Show which signals are passed to the subprocess with the minimum of `adb` interference. Signals may be added to or deleted from this list using the `:i` and `:t` commands.

`W` Re-open *objectfile* and *corefile* for writing, as though the `-w` command-line argument had been given.

`l` Sun386*i* only. Set the length in bytes (1, 2, or 4) of the object referenced by `:a` and `:w` to `address`. Default is 1.

8.6. adb Address Mapping

The interpretation of an address depends on its context. If a subprocess is being debugged, addresses are interpreted in the usual way (as described below) in the address space of the subprocess. If the operating system is being debugged, either post-mortem or by using the special file `/dev/mem` to examine and/or modify memory interactively, the maps are set to map the kernel virtual addresses, which start at zero. For some commands, the address is not interpreted as a memory address at all, but as an ordered pair representing a file number and a line number within that file. The `@` command always takes such a source file address, and several operators are available to convert to and from the more customary memory locations.

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*), and the *file address* corresponding to a written *address* is calculated as follows.

$$b1 \leq address < e1 \Rightarrow file\ address = address + f1 - b1$$

otherwise

$$\mathrm{b2} \leq \mathrm{address} < \mathrm{e2} \Rightarrow \mathrm{file\ address} = \mathrm{address} + \mathrm{f2} - \mathrm{b2}$$

Otherwise, the requested *address* is not legal. If a `?` or `/` request is followed by an `*`, only the second triple is used.

The initial setting of both mappings is suitable for normal object and core files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size, and *f1* is set to 0. This way, the whole file can be examined with no address translation.

8.7. See Also

For more information, read `dbx`(1), `ptrace`(2), `a.out`(5), and `core`(5) in the man pages.

8.8. Diagnostic Messages from adb

After startup, the only prompt `adb` gives is

```
adb
```

when there is no current command or format. On the other hand, `adb` supplies comments about inaccessible files, syntax errors, abnormal termination of commands, etc. The exit status is 0, unless the last command failed or returned nonzero status.

8.9. Bugs

There is no way to clear all breakpoints with a single command, except on the Sun386*i*.

Since no shell is invoked to interpret the arguments of the `:r` command, the customary wildcard and variable expansions cannot occur.

Since there is little type checking on addresses, using a source file address in an inappropriate context may lead to unexpected results.

8.10. Sun-3 FPA Support in `adb`

Release of the floating-point accelerator (FPA) for the Sun-3 required some changes to `adb`, in order to support assembly language debugging of programs that use the FPA.

1. The debugger variables `A` through `Z` are reserved for special use by `adb`. They should not be used in `adb` scripts.
2. The FPA registers `fpa0` through `fpa31` are recognized and can be used or modified in debugger commands. This extension only applies to systems with an FPA.
3. The debugger variable `F` governs FPA disassembly. This is equivalent to the `dbx` environment variable `fpaasm`. A value of 0 indicates that all FPA instructions are to be treated as move instructions. A nonzero value is used to indicate that FPA instruction sequences are to be disassembled and single stepped using FPA assembler mnemonics. On a machine with an FPA, the default value is 1; on other machines, the default value is 0.
4. The debugger variable `B` is used to designate an FPA base register. This is equivalent to the `dbx` environment variable `fpabase`. If FPA disassembly is disabled (the `F` flag = 0), its value is ignored. Otherwise, its value is interpreted as follows:

   0 through 7:
   : Based-mode FPA instructions that use the corresponding address register in `[a0..a7]` to address the FPA are also disassembled using FPA assembler mnemonics. Note that this is independent of the actual runtime value of the register.

   otherwise:
   : All based-mode FPA instructions are disassembled and single-stepped as move instructions.

   The default value of the FPA base register number is –1, which designates no FPA base register.
5. The command `$x` has been added to display the values of FPA registers `fpa0` through `fpa15`, along with FPA control registers and the current contents of the FPA instruction pipeline. All registers are displayed in the format:

```
<low word> <high word> <double precision> <single precision>
```

This verbose display is used because FPA registers are typeless; in particular, they may contain either single- or double-precision floating point values. If a single-precision value is stored, it is always stored in the high-order word. Machines without an FPA display the message "no FPA".

6. The command `$X` is similar to `$x`, but displays the FPA registers `fpa16` through `fpa31` instead of `fpa0` through `fpa15`. This is done as a separate command because `adb` cannot display the contents of all FPA registers in a single standard-size window.

7. The command `$R` displays the contents of the data and control registers of the standard MC68881 floating point coprocessor.

8.11. Examples of FPA Disassembly

As an example, consider the following assembly source fragment:

```
% cat foo.s
foo:
fpadds  d0,fpa0
fpadds@0    d0,fpa0
fpadds@5    d0,fpa0
%
```

On machines without an FPA, the default mode is to disassemble all FPA instructions as moves. For the example program, the following output is produced (except the parenthesized comments added here for explanation):

```
% as foo.s -o foo.o
% adb foo.o
<F=d
    0           (default value of 'F' on a machine without FPA)
foo?ia
foo:            movl    d0,0xe0000380    (normal disassembly)
```

FPA disassembly can be enabled by setting the debugger variable `F` to 1. For example:

```
% adb foo.o
1>F
<F=d
    1           (new value of 'F)
foo?ia
foo:            fpadds  d0,fpa0         (FPA disassembly)
```

On machines with an FPA, FPA disassembly is on by default, so the above output is produced without having to set the value of `F`.

Some FPA instructions may address the FPA using a base register in `[a0..a7]`. In practice, only `[a0..a5]` are used by the compilers.

`adb` does not know which register (if any) is being used to address the FPA in a given sequence of machine code. However, another debugger variable (`B`) may be set by the user to designate a register as an FPA base register. By default, this variable has the value –1, which means that no register should be assumed to point to the FPA, so only instructions that access the FPA using absolute addressing are recognized as FPA instructions.

For the example program, a machine with an FPA produces the following output:

```
% adb foo.o
<F=d
   1        (default value of 'F' on a machine with FPA)
<B=d
  -1        (default value of 'B')
foo,3?ia
foo:        fpadds  d0,fpa0      (FPA disassembly)
0x6:        movl    d0,a0@(0x380)    (normal disassembly)
0xa:        movl    d0,a5@(0x380)    (normal disassembly)
0xe:
```

Note that the second and third instructions are still disassembled as moves, since adb cannot assume that they access the FPA. Continuing this example, if the FPA base register number is set to 5, the following output is produced:

```
% adb foo.o
5>B
<B=d
   5
foo,3?ia
foo:        fpadds  d0,fpa0      (FPA disassembly)
0x6:        movl    d0,a0@(0x380)    (normal disassembly)
0xa:        fpadds@5    d0,fpa0  (FPA disassembly)
0xe:
```

Note that the second instruction is still disassembled as a move, since a5, the register designated as the FPA base, is not used in it.

FPA data registers can be displayed using a syntax similar to that used for the MC68881 co-processor registers. Note that unlike the MC68881 registers, FPA registers may contain either single-precision (32-bit) or double-precision (64-bit) values; MC68881 registers always contain an extended-precision (96-bit) value.

For example, if fpa0 contains the value 2.718282, we may display it as follows:

```
<fpa0=f
          fpa3     0x402df855    +2.718282e+00
```

Note that the value is displayed in hexadecimal as well as in floating-point notation. Unfortunately, an FPA register can only be set to a hexadecimal value. To set `fpa0` to 1.0, for example, you must know that this is represented as `0x3f800000` in IEEE single-precision format:

```
0x3f800000>fpa0
<fpa0=X
        3f800000
<fpa0=f
        +1.0000000e+00
```

9

Debugging SunOS Kernels with adb

This document describes the use of extensions made to the SunOS debugger `adb` for the purpose of debugging the SunOS kernel. It discusses the changes made to allow standard `adb` commands to function properly with the kernel and introduces the basics necessary for users to write `adb` command scripts that may be used to augment the standard `adb` command set. The examination techniques described here may be applied to running systems, as well as the post-mortem dumps automatically created by `savecore`(8) after a system crash. The reader is expected to have at least a passing familiarity with the debugger command language.

9.1. Introduction

Modifications have been made to the standard UNIX debugger `adb` to simplify examination of the post-mortem dump generated automatically following a system crash. These facilities may also be used when examining SunOS in its normal operation. This document serves as an introduction to the use of these facilities, but should not be construed as a description of how to debug the kernel.

Getting Started

Use the `-k` option of `adb` when you want to examine the SunOS kernel:

```
% adb -k /vmunix /dev/mem
```

The `-k` option makes `adb` partially simulate the Sun virtual memory management unit when accessing the *core* file. In addition, the internal state maintained by the debugger is initialized from data structures maintained by the SunOS kernel explicitly for debugging.† A post-mortem dump may be examined in a similar fashion:

```
% adb -k vmunix.? vmcore.?
```

Supply the appropriate version of the saved operating system image, and its core dump, in place of the question mark.

† If the `-k` flag is not used when invoking `adb`, the user must explicitly calculate virtual addresses. With the `-k` option, `adb` interprets page tables to perform virtual-to-physical address translation automatically.

Establishing Context

During initialization adb attempts to establish the context of the currently active process by examining the value of the kernel variable panic_regs. This structure contains the register values at the time of the call to the panic() routine. Once the stack pointer has been located, the following command generates a stack trace:

```
$c
```

An alternate method may be used when a trace of a particular process is required.

9.2. adb Command Scripts

This section supplies details about writing adb scripts to debug the kernel.

Extended Formatting Facilities

Once the process context has been established, the complete adb command set is available for interpreting data structures. In addition, a number of adb scripts have been created to simplify the structured printing of commonly referenced kernel data structures. The scripts normally reside in the directory /usr/lib/adb, and are invoked with the $< operator. Standard scripts are listed below in Table 8-1.

As an example, consider the listing that starts below. The listing contains a dump of a faulty process's state.

```
% adb -k vmunix.3 vmcore.3
sbr 50030 slr 51e
physmem 3c0
$c
_panic[10fec](5234d) + 3c
_ialloc[16ea8](d44a2,2,dff) + c8
_maknode[1d476](dff) + 44
_copen[1c480](602,-1) + 4e
_creat() + 16
_syscall[2ea0a]() + 15e
level5() + 6c
5234d/s
_nldisp+175:    ialloc: dup alloc
u$<u
_u:
_u:     pc
        4be0
_u+4:       d2      d3      d4      d5
        13b0        0       0       0
_u+14:      d6      d7
        0       2604
_u+1c:      a2      a3      a4      a5
        0       c7800       5a958       d7160
_u+2c:      a6      a7
        3e62        3e48
```

```
_u+34:       sr
        27000000
_u+38:       p0br          p0lr          p1br          p1lr
        105000        40000022      fd7f4         1ffe
_u+48:       szpt          sswap
        1       0
_u+50:       procp         ar0      comm
        d7160         3fb2          dtime^@^@^@^@^@
_u+158:      arg0          arg1          arg2
        1001c         -1       ffffa4
_u+178:      uap      qsave                    error
        2958          2eb46         1        0
_u+1b2:      rv1      rv2      eosys
        0       14cac         0
_u+1bc:      uid gid
        49  10
_u+1c0:      groups
        10      -1      -1      -1
        -1      -1      -1      -1
_u+1e0:      ruid     rgid
        49  10
_u+1e4:      tsize         dsize         ssize
        7       1b      2
_u+344:      odsize        ossize        outime
        0       0       0
_u+350:      signal
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        sigmask
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
_u+450:      onstack       oldmask       code
        0       80002         0
_u+45c:      sigstack      onsigstack
        0       0
```

```
_u+464:     ofile
        d66b4       d66b4       d66b4       0
        0       0       0       0
        0       0       0       0
        0       0       0       0
        0       0       0       0

        pofile
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0
_u+4c8:     cdir    rdir    ttyp    ttyd    cmask
        d44a2       0       5c6c0       0   12

        ru & cru
_u+4d8:     utime               stime
        0       0       0       35b60
_u+4e8:     maxrss      ixrss       idrss       isrss
        9       35      43
_u+4f8:     minflt      majflt      nswap
        0       5       0
_u+504:     inblock     oublock     msgsnd      msgrcv
        3       7       0       0
_u+514:     nsignals    nvcsw       nivcsw
        0       12      4
_u+520:     utime               stime
        0       0       0       0
_u+530:     maxrss      ixrss       idrss       isrss
        0       0       0
_u+540:     minflt      majflt      nswap
        0       0       0
_u+54c:     inblock     oublock     msgsnd      msgrcv
        0       0       0       0
_u+55c:     nsignals    nvcsw       nivcsw
        0       0       0
0d7160$<proc
d7160:      link        rlink       addr
        590e0       0       1057f4
d716c:      upri    pri cpu stat    time    nice    slp
        066 024 020 03  01  024 0
d7173:      cursig      sig
        0       0
d7178:      mask        ignore      catch
        0       0       0
d7184:      flag        uid pgrp    pid ppid
        8001        31  2f  2f  23
d7190:      xstat       ru      poip    szpt    tsize
        0       0       0   1   7
d719e:      dsize       ssize       rssize      maxrss
        1b      2       5       fffff
```

```
d71ae:          swrss           swaddr          wchan           textp
        0       0               0       d8418
d71be:          p0br            xlink           ticks
        105000          0       15
d71c8:          %cpu                    ndx idhash      pptr
        0                       6       2       d70d4
d71d4:          real itimer
        0       0               0       0
d71e4:          quota           ctx
        0       5f236
0d8418$<text
d8418:          daddr
    284         0       0       0
    0           0       0       0
    0           0       0       0

    ptdaddr         size            caddr           iptr
    184         7       d7160           d47e0

    rssize  swrss   count   ccount  flag    slptim  poip
    4   0   01  01  042 0   0
```

The cause of the crash was a `panic` (see the stack trace) due to a duplicate `inode` allocation detected by the `ialloc()` routine. The majority of the dump was done to illustrate the use of command scripts used to format kernel data structures. The `u` script, invoked by the command `u$<u`, is a lengthy series of commands to pretty-print the user vector. Likewise, `proc` and `text` are scripts to format the obvious data structures. Let's quickly examine the `text` script, which has been broken into a number of lines for readability here; in actuality it is a single line of text.

```
./"daddr"n12Xn\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

The first line produces the list of disk block addresses associated with a swapped out text segment. The `n` format forces a newline character, with 12 hexadecimal integers printed immediately after. Likewise, the remaining two lines of the command format the remainder of the text structure. The expression `16t` tabs to the next column which is a multiple of 16.

The majority of the scripts provided are of this nature. When possible, the formatting scripts print a data structure with a single format to allow subsequent reuse when interrogating arrays of structures. That is, the previous script could have been written:

```
./"daddr"n12Xn
+/"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn
+/"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

But then, reuse of the format would have invoked only the last line of the format.

Traversing Data Structures

The adb command language can be used to traverse complex data structures. One such data structure, a linked list, occurs quite often in the kernel. By using adb variables and the normal expression operators it is a simple matter to construct a script which chains down the list, printing each element along the way.

For instance, the queue of processes awaiting timer events, the callout queue, is printed with the following two scripts:

```
callout:
    calltodo/"time"16t"arg"16t"func"
    *(.+0t12)$<callout.nxt
```

```
callout.nxt:
    ./D2p
    *+>l
    ,#<l$<
    <l$<callout.nxt
```

The first line of the script callout starts the traversal at the global symbol calltodo and prints a set of headings. It then skips the empty portion of the structure used as the head of the queue. The second line then invokes the script callout.nxt moving *dot* to the top of the queue — *+ performs the indirection through the link entry of the structure at the head of the queue. The script callout.nxt prints values for each column, then performs a conditional test on the link to the next entry. This test is performed as follows:

```
*+>l
```

This means to place the value of the *link* in the adb variable <l. Next:

```
,#<l$<
```

This means if the value stored in <l is non-zero, then the current input stream (from the script callout.nxt) is terminated. Otherwise, the expression #<l is zero, and the $< operator is ignored. That is, the combination of the logical negation operator #, adb variable <l, and operator $<, in effect, creates a statement of the form:

```
if (!link)
    exit;
```

The remaining line of callout.nxt simply reapplies the script on the next element in the linked list. A sample callout dump is shown below:

```
% adb -k /vmunix /dev/mem
sbr 50030 slr 51e
physmem 3c0
$<callout
_calltodo:
_calltodo:  time        arg     func
d9fc4:      5       0       _roundrobin
d9f94:      1       0       _if_slowtimo
d9fd4:      1       0       _schedcpu
d9fa4:      3       0       _pffasttimo
d9fe4:      0       0       _schedpaging
d9fb4:      15      0       _pfslowtimo
d9ff4:      12      0       _arptimer
da044:      736     d7390       _realitexpire
da004:      206     d6fbc       _realitexpire
da024:      649     d741c       _realitexpire
da034:      176929      d7304       _realitexpire
```

Supplying Parameters

A command script may use the address and count portions of an adb command as parameters. An example of this is the setproc script, used to switch to the context of a process with a known process ID:

```
0t99$<setproc
```

The body of setproc is:

```
.>4
*nproc>l
*proc>f
$<setproc.nxt
```

The body of setproc.nxt is:

```
(*(<f+0t42)&0xffff)="pid "D
,#(((*(<f+0t42)&0xffff))-<4)$<setproc.done
<l-1>l
<f+0t140>f
,#<l$<
$<setproc.nxt
```

The process ID, supplied as the parameter, is stored in the variable <4, the number of processes is placed in <l, and the base of the array of process structures in <f. Then setproc.nxt performs a linear search through the array until it matches the process ID requested, or until it runs out of process structures to check. The script setproc.done simply establishes the context of the process, then exits.

Standard Scripts

Here are the command scripts currently available in `/usr/lib/adb`:

Table 9-1 *Standard Command Scripts*

Name	*Use*	*Description*
`buf`	*addr*`$<buf`	format block I/O buffer
`callout`	`$<callout`	print timer queue
`clist`	*addr*`$<clist`	format character I/O linked list
`dino`	*addr*`$<dino`	format directory inode
`dir`	*addr*`$<dir`	format directory entry
`file`	*addr*`$<file`	format open file structure
`filsys`	*addr*`$<filsys`	format in-core super block structure
`findproc`	*pid*`$<findproc`	find process by process id
`ifnet`	*addr*`$<ifnet`	format network interface structure
`inode`	*addr*`$<inode`	format in-core inode structure
`inpcb`	*addr*`$<inpcb`	format internet protocol control block
`iovec`	*addr*`$<iovec`	format a list of *iov* structures
`ipreass`	*addr*`$<ipreass`	format an ip reassembly queue
`mact`	*addr*`$<mact`	show active list of mbufs
`mbstat`	`$<mbstat`	show mbuf statistics
`mbuf`	*addr*`$<mbuf`	show next list of mbufs
`mbufs`	*addr*`$<mbufs`	show a number of mbufs
`mount`	*addr*`$<mount`	format mount structure
`pcb`	*addr*`$<pcb`	format process context block
`proc`	*addr*`$<proc`	format process table entry
`protosw`	*addr*`$<protosw`	format protocol table entry
`rawcb`	*addr*`$<rawcb`	format a raw protocol control block
`rtentry`	*addr*`$<rtentry`	format a routing table entry
`rusage`	*addr*`$<rusage`	format resource usage block
`setproc`	*pid*`$<setproc`	switch process context to *pid*
`socket`	*addr*`$<socket`	format socket structure
`stat`	*addr*`$<stat`	format stat structure
`tcpcb`	*addr*`$<tcpcb`	format TCP control block
`tcpip`	*addr*`$<tcpip`	format a TCP/IP packet header
`tcpreass`	*addr*`$<tcpreass`	show a TCP reassembly queue
`text`	*addr*`$<text`	format text structure
`traceall`	`$<traceall`	show stack trace for all processes
`tty`	*addr*`$<tty`	format tty structure
`u`	*addr*`$<u`	format user vector, including pcb
`uio`	*addr*`$<uio`	format uio structure
`vtimes`	*addr*`$<vtimes`	format vtimes structure

9.3. Generating adb Scripts with adbgen

You can use the adbgen program to write the scripts presented earlier in a way that does not depend on the structure member offsets of referenced items. For example, the text script given above depends on all printed members being located contiguously in memory. Using adbgen, the script could be written as follows (again it is really on one line, but broken apart for ease of display):

```
#include "sys/types.h"
#include "sys/text.h"

text
./"daddr"n{x_daddr,12X}n\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n\
{x_ptdaddr,X}{x_size,X}{x_caddr,X}{x_iptr,X}n\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n\
{x_rssize,x}{x_swrss,x}{x_count,b}{x_ccount,b}\
{x_flag,b}{x_slptime,b}{x_poip,x}{END}
```

The script starts with the names of the relevant header files, while the braces delimit structure member names and their formats. This script is then processed through adbgen to get the adb script presented in the previous section. See Chapter 10 of this manual for a complete description of how to write adbgen scripts. The real value of writing scripts this way becomes apparent only with longer and more complicated scripts (the u script for example). When scripts are written this way, they can be regenerated if a structure definition changes, without requiring that the offsets be recalculated.

10

Generating adb Scripts with adbgen

/usr/lib/adb/adbgen *file*.adb ...

This program makes it possible to write adb scripts that do not contain hard-coded dependencies on structure member offsets. After generating a C program to determine structure member offsets and sizes, adbgen proceeds to generate an adb script.

The input to adbgen is a file named *file*.adb containing adbgen header information, then a null line, then the name of a structure, and finally an adb script. The adbgen program only deals with one structure per file; all member names occurring in a file are assumed to be in this structure. The output of adbgen is an adb script in *file* (without the .adb suffix).

The header lines, up to the null line, are copied verbatim into the generated C program. These header lines often have #include statements to read in header files containing relevant structure declarations.

The second part of *file*.adb specifies a structure.

The third part contains an adb script with any valid adb commands (see Chapter 6 of this manual), and may also contain adbgen requests, each enclosed in braces. Request types are:

1) Print a structure member. The request form is {*member*,*format*} where *member* is a member name of the structure given earlier, and *format* is any valid adb format request. For example, to print the p_pid field of the proc structure as a decimal number, say {p_pid,d}.

2) Reference a structure member. The request form is {**member*,*base*} where *member* is the member name whose value is wanted, and *base* is an adb register name containing the base address of the structure. For example, to get the p_pid field of the proc structure, get the proc structure address in an adb register, such as <f, and say {*p_pid,<f}.

3) Tell adbgen that the offset is OK. The request form is {OFFSETOK}. This is useful after invoking another adb script which moves the adb *dot*.

4) Get the size of the *structure*. The request form is {SIZEOF}; adbgen simply replaces this request with the size of the structure. This is useful for incrementing a pointer to step through an array of structures.

5) Get the offset to the end of the structure. The request form is {END}. This is useful at the end of a structure to get adb to align *dot* for printing the next structure member.

By keeping track of the movement of *dot*, adbgen emits adb code to move forward or backward as necessary before printing any structure member in a script. The model of *dot*'s behavior is simple: adbgen assumes that the first line of the script is of the form *struct_address*/ *adb text* and that subsequent lines are of the form +/ *adb text*. This causes *dot* to move in a sane fashion. Unfortunately, adbgen does not check the script to ensure that these limitations are met. However, adbgen does check the size of the structure member against the size of the adb format code, and warns you if they are not equal.

10.1. Example of adbgen

If there were an include file x.h like this,

```
struct x {
    char    *x_cp;
    char    x_c;
    int x_i;
};
```

then the adbgen file (call it script.adb) to print it would be:

```
#include "x.h"

x
./"x_cp"16t"x_c"8t"x_i"n{x_cp,X}{x_c,C}{x_i,D}
```

After running adbgen, the output file script would contain:

```
./"x_cp"16t"x_c"8t"x_i"nXC+D
```

To invoke the script, type:

```
x$<script
```

10.2. Diagnostic Messages from adbgen

The adbgen program generates warnings about structure member sizes not equal to adb format items, and complaints about badly formatted requests. The C compiler complains if you reference a nonexistent structure member. It also complains about & before array names; these complaints may be ignored.

10.3. Bugs in adbgen

Structure members that are bit fields cannot be handled, because C will not give the address of a bit field; the address is needed to determine the offset.

Index

V

W

Notes

Notes