Digital UNIX

Kernel Debugging

Part Number: AA-PS2TE-TE

March 1996

Product Version: Digital UNIX Version 4.0 or higher

This manual explains how to use tools to debug a kernel and analyze a crash dump of the Digital UNIX operating system. Also, this manual explains how to write extensions to the kernel debugging tools.

Digital Equipment Corporation
Maynard, Massachusetts

Contents

3 Writing Extensions to the kdbx Debugger

4 Managing Crash Dumps

5 Crash Analysis Examples

A Output from the crashdc Command

Index

Examples

Figures

Tables

About This Manual

This manual provides information on the tools used to debug a kernel and analyze a crash dump file of the Digital UNIX operating system. It also explains how to write extensions to the kernel debugging tools. You can use extensions to display customized information from kernel data structures or a crash dump file.

Audience

This manual is intended for system programmers who write programs that use kernel data structures and are built into the kernel. It is also intended for system administrators who are responsible for managing the operating system. System programmers and administrators should have in-depth knowledge of operating system concepts, commands, and utilities.

New and Changed Features

The following list describes changes that have been made to this manual for Digital UNIX Version 4.0 or higher:

- Section 1.1, describing how to debug a kernel that is linked at bootstrap time, has been added.
- Section 2.1.2, describing how to debug images that are stripped of symbol table information, has been added.
- The new `abscallout kdbx` extension is explained in Section 2.2.3.4.
- Example 3–2, Example 3–4, andExample 3–5 have been updated.

Organization

This manual consists of five chapters and one appendix:

Chapter 1	Introduces the concepts of kernel debugging and crash dump analysis.
Chapter 2	Describes the tools used to debug kernels and analyze crash dump files.
Chapter 3	Describes how to write a kdbx debugger extension. This chapter assumes you have purchased and installed a Digital UNIX Source Kit and so have access to source files.
Chapter 4	Describes the crash dump file creation process and explains how to manage crash dump files on your system.
Chapter 5	Provides background information useful for and examples of analyzing crash dump files.
Appendix A	Contains example output from the crashdc utility.

Related Documents

For additional information, refer to the following manuals:

- The *Alpha Architecture Reference Manual* describes how the operating system interfaces with the Alpha hardware.
- The *Alpha Architecture Handbook* gives an overview of the Alpha hardware architecture and describes the 64-bit Alpha RISC (Reduced Instruction Set Computing) instruction set.
- The *Installation Guide* describes how to install your operating system.
- The *System Administration* manual provides information on managing and monitoring your system.
- The *Programmer's Guide* provides information on the tools, specifically the dbx debugger, for programming on the Digital UNIX operating system. This manual also provides information about creating configurable kernel subsystems.

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

Reader's Comments

Digital welcomes any comments and suggestions you have on this and other Digital UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on line in the following location:

  `/usr/doc/readers_comment.txt`

- Mail:

  Digital Equipment Corporation
  UEG Publications Manager
  ZK03-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-9987

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using.
- If known, the type of processor that is running the Digital UNIX software.

The Digital UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

Conventions

The following conventions are used in this manual:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
`#`	A number sign represents the superuser prompt.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.

`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

1

Introduction to Kernel Debugging

Kernel debugging is a task normally performed by systems engineers writing kernel programs. A kernel program is one that is built as part of the kernel and that references kernel data structures. System administrators might also debug the kernel in the following situations:

- A process is hung or stops running unexpectedly
- The need arises to examine, and possibly modify, kernel parameters
- The system itself hangs, panics, or crashes

This manual describes how to use Digital UNIX tools to debug kernel programs and the kernel. It also includes information about managing and analyzing crash dump files.

In addition to the information provided here, tracing a kernel problem can require a basic understanding of one or more of the following technical areas:

- The hardware architecture

  See the *Alpha Architecture Handbook* for an overview of the Alpha hardware architecture and a description of the 64-bit Alpha RISC instruction set.

- The internal design of the operating system at a source code and data structure level

  See the *Alpha Architecture Reference Manual* for information on how the Digital UNIX operating system interfaces with the hardware.

This chapter provides an overview of the following topics:

- Linking a kernel image prior to debugging for systems that are running a kernel built at boot time. (Section 1.1)
- Debugging kernel programs (Section 1.2)
- Debugging the running kernel (Section 1.3)

- Analyzing a crash dump file(Section 1.4)

1.1 Linking a Kernel Image for Debugging

By default, the kernel that runs on Digital UNIX systems is a statically linked image that resides in the file /vmunix. However, your system might be configured so that it is linked at bootstrap time. Rather than being a bootable image, the boot file is a text file that describes the hardware and software that will be present on the running system. Using this information, the bootstrap linker links the modules that are needed to support this hardware and software. The linker builds the kernel directly into memory. (For more information about bootstrap-linked kernels, see the manual *Writing Device Drivers: Tutorial*.)

You cannot directly debug a bootstrap-linked kernel because you must supply the name of an image to the kernel debugging tools. Without the image, the tools have no access to symbol names, variable names, and so on. Therefore, the first step in any kernel debugging effort is to determine whether your kernel was linked at bootstrap time. If the kernel was linked at bootstrap time, you must then build a kernel image file to use for debugging purposes.

The best way to determine whether your system is bootstrap linked or statically linked is to use the `file` command to test the type of file from which your system was booted. If your system is a bootstrap-linked system, it was booted from an ASCII text file; otherwise, it was booted from an executable image file. For example, issue the following command to determine the type of file from which your system was booted:

```
#/usr/bin/file '/usr/sbin/sizer -b'
/etc/sysconfigtab: ascii text
```

The `sizer -b` command returns the name of the file from which the system was booted. This file name is input to the `file` command, which determines that the system was booted from an ASCII text file. The output shown in the preceeding example indicates that the system is a bootstrap-linked system. If the system had been booted from an executable image file named `vmunix`, the output from the `file` command would have appeared as follows:

```
vmunix:COFF format alpha executable or object module
 not stripped
```

If your system is running a bootstrap-linked kernel, build a kernel image that is identical to the bootstrap-linked kernel your system is running, by entering the following command:

```
# /usr/bin/ld -o vmunix.image ‘/usr/sbin/sizer -m‘
```

The output from the `sizer -m` command is a list of the exact modules and linker flags used to build the currently running bootstrap-linked kernel. This output causes the `ld` command to create a kernel image that is identical to the bootstrap-linked kernel running on your system. The kernel image is written to the file named by the `-o` flag, in this case the `vmunix.image` file.

Once you create this image, you can debug the kernel as described in this manual, using the `dbx`, `kdbx`, and `kdebug` debuggers. When you invoke the `dbx` or `kdbx` debugger, remember to specify the name of the kernel image file you created with the `ld` command, such as the `vmunix.image` file shown here.

When you are finished debugging the kernel, you can remove the kernel image file you created for debugging purposes.

1.2 Debugging Kernel Programs

Kernel programs can be difficult to debug because you normally cannot control kernel execution. To make debugging kernel programs more convenient, the Digital UNIX system provides the `kdebug` debugger. The `kdebug` debugger is code that resides inside the kernel and allows you use the `dbx` debugger to control execution of a running kernel in the same manner as you control execution of a user space program. To debug a kernel program in this manner, follow these steps:

1. Build your kernel program into the kernel on a test system.
2. Set up the `kdebug` debugger, as described in Section 2.3.
3. Issue the `dbx -remote` command on a remote build system, supplying the pathname of the kernel running on the test system.
4. Set breakpoints and enter `dbx` commands as you normally would. Section 2.1 describes some of the commands that are useful during kernel debugging. For general information about using `dbx`, see the *Programmer's Guide*.

The Digital UNIX system also provides the kdbx debugger, which is designed especially for debugging kernel code. This debugger contains a number of special commands, called extensions, that allow you to display kernel data structures in a readable format. Section 2.2 describes using kdbx and its extensions. (You cannot use the kdbx debugger with the kdebug debugger.)

Another feature of kdbx is that you can customize it by writing your own extensions. The system contains a set of kdbx library routines that you can use to create extensions that display kernel data structures in ways that are meaningful to you. Chapter 3 describes writing kdbx extensions.

1.3 Debugging the Running Kernel

When you have problems with a process or set of processes, you can attempt to identify the problem by debugging the running kernel. You might also invoke the debugger on the running kernel to examine the values assigned to system parameters. (You can modify the value of the parameters using the debugger, but this practice can cause problems with the kernel and should be avoided.)

You use the dbx or kdbx debugger to examine the state of processes running on your system and to examine the value of system parameters. The kdbx debugger provides special commands, called extensions, that you can use to display kernel data structures. (Section 2.2.3 describes the extensions.)

To examine the state of processes, you invoke the debugger (as described in Section 2.1 or Section 2.2) using the following command:

```
# dbx -k /vmunix /dev/mem
```

This command invokes dbx with the kernel debugging flag, −k, which maps kernel addresses to make kernel debugging easier. The /vmunix and /dev/mem parameters cause the debugger to operate on the running kernel.

Once in the dbx environment, you use dbx commands to display process IDs and trace execution of processes. You can perform the same tasks using the kdbx debugger. The following example shows the dbx command you use to display process IDs:

```
(dbx) kps
  PID   COMM
00000   kernel idle
```

```
00001   init
00014   kloadsrv
00016   update
   .
   .
   .
```

If you want to trace the execution of the kloadsrv daemon, use the dbx command to set the $pid symbol to the process ID of the kloadsrv daemon. Then, enter the t command:

```
(dbx) set $pid = 14
(dbx) t
>  0 thread_block() ["/usr/sde/build/src/kernel/kern/sched_prim.c":1623, 0xfffffc0000\
43d77c]
   1 mpsleep(0xffffffff92586f00, 0x11a, 0xfffffc0000279cf4, 0x0, 0x0) ["/usr/sde/build\
/src/kernel/bsd/kern_synch.c":411, 0xfffffc000040adc0]
   2 sosleep(0xffffffff92586f00, 0x1, 0xfffffc000000011a, 0x0, 0xffffffff81274210) ["/usr/sde\
/build/src/kernel/bsd/uipc_socket2.c":654, 0xfffffc0000254ff8]
   3 sosbwait(0xffffffff92586f60, 0xffffffff92586f00, 0x0, 0xffffffff92586f00, 0x10180) ["/usr\
/sde/build/src/kernel/bsd/uipc_socket2.c":630, 0xfffffc0000254f64]
   4 soreceive(0x0, 0xffffffff9a64f658, 0xffffffff9a64f680, 0x8000004300000000, 0x0) ["/usr/sde\
/build/src/kernel/bsd/uipc_socket.c":1297, 0xfffffc0000253338]
   5 recvit(0xfffffc0000456fe8, 0xffffffff9a64f718, 0x14000c6d8, 0xffffffff9a64f8b8,\
 0xfffffc000043d724) ["/usr/sde/build/src/kernel/bsd/uipc_syscalls.c":1002,\
 0xfffffc00002574f0]
   6 recvfrom(0xffffffff81274210, 0xffffffff9a64f8c8, 0xffffffff9a64f8b8, 0xffffffff9a64f8c8,\
 0xfffffc0000457570) ["/usr/sde/build/src/kernel/bsd/uipc_syscalls.c":860,\
 0xfffffc000025712c]
   7 orecvfrom(0xffffffff9a64f8b8, 0xffffffff9a64f8c8, 0xfffffc0000457570, 0x1, 0xfffffc0000456fe8)\
 ["/usr/sde/build/src/kernel/bsd/uipc_syscalls.c":825, 0xfffffc000025708c]
   8 syscall(0x120024078, 0xffffffffffffffff, 0xffffffffffffffff, 0x21, 0x7d) ["/usr/sde\
/build/src/kernel/arch/alpha/syscall_trap.c":515, 0xfffffc0000456fe4
   9 _Xsyscall(0x8, 0x12001acb8, 0x14000eed0, 0x4, 0x1400109d0) ["/usr/sde/build\
/src/kernel/arch/alpha/locore.s":1046, 0xfffffc00004486e4]
(dbx) exit
```

Often, looking at the trace of a process that is hanging or has unexpectedly stopped running reveals the problem. Once you find the problem, you can modify system parameters, restart daemons, or take other corrective actions.

For more information about the commands you can use to debug the running kernel, see Section 2.1 and Section 2.2.

1.4 Analyzing a Crash Dump File

If your system crashes, you can often find the cause of the crash by using dbx or kdbx to debug or analyze a crash dump file.

The operating system can crash because one of the following occurs:

- Hardware exception
- Software panic

- Hung system

  When a system hangs, it is often necessary to force the system to create dumps that you can analyze to determine why the system hung. Section 4.7 describes the procedure for forcing a crash dump of a hung system.

- Resource exhaustion

The system crashes or hangs because it cannot continue executing. Normally, even in the case of a hardware exception, the operating system detects the problem. (For example a machine-checking routine might discover a hardware problem and begin the process of crashing the system.) In general, the operating system performs the following steps when it detects a problem from which it cannot recover:

1. Calls the system `panic` function.

   The `panic` function saves the contents of registers and sends the panic string (a message describing the reason for the system panic) to the error logger and the console terminal.

   If the system is a Symmetric Multiprocessing (SMP) system, the `panic` function notifies the other CPUs in the system that a `panic` has occurred. The other CPUs then also execute the `panic` function and record the following panic string:

   ```
   cpu_ip_intr: panic request
   ```

   Once each CPU has recorded the system panic, execution continues only on the master CPU. All other CPUs in the SMP system stop execution.

2. Calls the system `boot` function

   The `boot` function records the stack.

3. Calls the `dump` function

   The `dump` function copies core memory into swap partitions and the system stops running or the reboot process begins. Console environment variables control whether the system reboots automatically. (The *Installation Guide* describes these environment variables.)

At system reboot time, the copy of core memory saved in the swap partitions is copied into a file, called a crash dump file. You can analyze the crash dump file to determine what caused the crash. For information about managing crash dumps and crash dump files, see Chapter 4. For examples of analyzing crash dump files, see Chapter 5.

2

Kernel Debugging Utilities

The Digital UNIX system provides several tools you can use to debug the kernel and kernel programs. The Ladebug debugger (available separately from the Digital UNIX operating system) is also capable of debugging the kernel. For information about the Ladebug debugger, contact your Digital sales representative.

This chapter describes three debuggers and one crash dump analysis utility:

- The `dbx` debugger, which is described for kernel debugging in Section 2.1. (For general `dbx` user information, see the *Programmer's Guide*.)

  You can use the `dbx` debugger to display the values of kernel variables and kernel structures. However, you must understand the structures and be prepared to follow the address links to find the information you need. You cannot use `dbx` alone to control execution of the running kernel, for example by setting breakpoints.

- The `kdbx` debugger, which is described in Section 2.2.

  The `kdbx` debugger is an interface to `dbx` that is tailored specifically to debugging kernel code. The `kdbx` debugger has knowledge of the structure of kernel data and so displays kernel data in a readable format. Also, `kdbx` is extensible, allowing you to create commands that are tailored to your kernel-debugging needs. (Chapter 3 describes how to tailor the `kdbx` debugger.) However, you cannot use `dbx` command line editing features when you use the `kdbx` debugger.

- The `kdebug` debugger, which is described in Section 2.3.

  The `kdebug` debugger is a kernel-debugging program that resides inside the kernel. Working with a remote version of the `dbx` debugger, the `kdebug` debugger allows you to set breakpoints in and control the execution of kernel programs and the kernel.

- The `crashdc` utility, which is described in Section 2.4.

The crashdc utility is a crash dump analysis tool. This utility is useful when you need to determine why the system is hanging or crashing.

The sections that follow describe how to use these tools to debug the kernel and kernel programs.

2.1 The dbx Debugger

The dbx debugger is a symbolic debugger that allows you to examine, modify, and display the variables and data structures found in stripped or nonstripped kernel images.

The following sections describe how to invoke the dbx debugger for kernel debugging and how to use its commands to perform tasks such as the following:

- Debugging stripped images
- Examining memory contents
- Displaying the values of kernel variables, and the value and format of kernel data structures
- Debugging multiple threads

Also included are examples of examining the exception frame and the preserved character message buffer. For more information on dbx, see the *Programmer's Guide*.

2.1.1 Kernel Debugging Flag

To debug kernel code with the dbx debugger, you use the –k flag. This flag causes dbx to map memory addresses. When you use the dbx –k command, the debugger operates on two separate files that reflect the current state of the kernel that you want to examine. These files are as follows:

- The disk version of the executable kernel image
- The system core memory image

These files may be files from a running system, such as /vmunix and /dev/mem, or dump files, such as vmunix.*n* and vmcore.*n*, which usually reside in the /var/adm/crash directory.

Note

You might need to be the superuser (root login) to examine the running system or crash dump files produced by savecore. Whether you need to be the superuser depends on the directory and file protections for the files you attempt to examine with the dbx debugger.

Use the following dbx command to examine the running system:

```
# dbx –k /vmunix /dev/mem
```

Use the following command to examine crash dump files with bounds equal to one:

```
# dbx –k vmunix.1 vmcore.1
```

2.1.2 Debugging Stripped Images

By default, the kernel is compiled with a debugging flag that does not strip all of the symbol table information from the executable kernel image. The kernel is also partially optimized during the compilation process by default. If the kernel or any other file is fully optimized and stripped of all symbol table information during compilation, your ability to debug the file is greatly reduced. However, the dbx debugger provides commands to aid you in debugging stripped images.

When you attempt to display the contents of a symbol during a debugging session, you might encounter messages such as the following:

```
No local symbols.
Undefined symbol.
Inactive symbol.
```

These messages might indicate that you are debugging a stripped image.

To see the contents of all symbols during a debugging session, you can leave the debugging session, rebuild all stripped modules (but do not strip them), and reenter the debugging session. However, on certain occasions, you might want to add a symbol table to your current debugging session rather than end the session and start a new one. To add a symbol table to your current debugging session, follow these steps:

1. Go to a window other than the one in which the debugger is running, or put the debugger in the background, and rebuild the modules for which you need a symbol table.
2. Once the modules build correctly, use the `ostrip` command to strip a symbol table out of the resulting executable file. For example, if your executable file is named `kernel_program`, issue a command such as the following one:

   ```
   % /usr/ucb/ostrip -t kernel_program
   ```

   The `-t` flag causes the `ostrip` command to produce two files. One, named `kernel_program`, is the stripped executable image. The other, named `kernel_program.stb`, contains the symbol table information for the `kernel_program` module. (For more information about the `ostrip` command, see `ostrip`(1).)
3. Return to the debugging session and add the symbol table file by issuing the `dbx` command `stbadd` as follows:

   ```
   dbx> stbadd kernel_program.stb
   ```

   You can specify an absolute or relative pathname on the `stbadd` command line.

   Once you issue this command, you can display the contents of symbols included in the symbol table just as if you had built the module you are debugging without stripping.

You can also delete symbol tables from a debugging session using the `dbx` command `stbdel`. For more information about this command, see `dbx`(1).

2.1.3 Examining Memory Contents

To examine memory contents with `dbx`, use the following syntax:

```
address/count[mode]
```

The *count* argument specifies the number of items that the debugger displays at the specified *address*, and the *mode* argument determines how `dbx` displays memory. If you omit the *mode* argument, the debugger uses the previous mode. The initial default mode is X (hexadecimal). Table 2–1 lists the dbx address modes.

Table 2–1: The dbx Address Modes

Mode	Description
b	Displays a byte in octal.
c	Displays a byte as a character.
d	Displays a short word in decimal.
D	Displays a long word in decimal.
f	Displays a single precision real number.
g	Displays a double precision real number.
i	Displays machine instructions.
n	Displays data in typed format.
o	Displays a short word in octal.
O	Displays a long word in octal.
s	Displays a string of characters that ends in a null.
x	Displays a short word in hexadecimal.
X	Displays a long word in hexadecimal.

The following examples show how to use dbx to examine kernel images:

```
(dbx) _realstart/X
fffffc00002a4008:  c020000243c4153e
(dbx) _realstart/i
[_realstart:153, 0xfffffc00002a4008]  subq    sp, 0x20, sp
(dbx) _realstart/10i
  [_realstart:153, 0xfffffc00002a4008]  subq    sp, 0x20, sp
  [_realstart:154, 0xfffffc00002a400c]  br      r1, 0xfffffc00002a4018
  [_realstart:156, 0xfffffc00002a4010]  call_pal        0x4994e0
  [_realstart:157, 0xfffffc00002a4014]  bgt     r31, 0xfffffc00002a3018
  [_realstart:171, 0xfffffc00002a4018]  ldq     gp, 0(r1)
  [_realstart:172, 0xfffffc00002a401c]  stq     r31, 24(sp)
  [_realstart:177, 0xfffffc00002a4020]  bis     r16, r31, r9
  [_realstart:178, 0xfffffc00002a4024]  bis     r17, r31, r10
  [_realstart:179, 0xfffffc00002a4028]  bis     r18, r31, r11
  [_realstart:181, 0xfffffc00002a402c]  bis     r19, r31, r12
```

2.1.4 Printing the Values of Variables and Data Structures

You can use the `print` command to examine values of variables and data structures. The `print` command has the following syntax:

print *expression*

p *expression*

For example:

```
(dbx) print utsname
struct {
    sysname = "OSF1"
    nodename = "system.dec.com"
    release = "1.4"
    version = "1.2"
    machine = "alpha"
}
```

Note that `dbx` has a default alias of `p` for `print`:

```
(dbx) p utsname
```

2.1.5 Displaying a Data Structure Format

You can use the `whatis` command to display the format for many of the kernel data structures. The `whatis` command has the following syntax:

whatis *type name*

The following example displays the `itimerval` data structure:

```
(dbx) whatis struct itimerval
struct itimerval {
    struct timeval {
        int tv_sec;
        int tv_usec;
    } it_interval;
    struct timeval {
        int tv_sec;
        int tv_usec;
    } it_value;
};
```

2.1.6 Debugging Multiple Threads

You can use the dbx debugger to examine the state of the kernel's threads with the querying and scoping commands described in this section. You use these commands to show process and thread lists and to change the debugger's context (by setting its current process and thread variables) so that a stack trace for a particular thread can be displayed. Use these commands to examine the state of the kernel's threads:

`print $tid`	Display the thread ID of the current thread
`print $pid`	Display the process ID of the current process
`trace`	Display a stack trace for the current thread
`tlist`	Display a list of kernel threads for the current process
`kps`	Display a list of processes (not available when used with kdebug)
`set $pid=`*process_id*	Change the context to another process (a process ID of 0 changes context to the kernel)
`tset` *thread_id*	Change the context to another thread
`tstack`	Displays the stack trace for all threads.

2.1.7 Examining the Exception Frame

When you work with a crash dump file to debug your code, you can use dbx to examine the exception frame. The exception frame is a stack frame created during an exception. It contains the registers that define the state of the routine that was running at the time of the exception. Refer to the /usr/include/machine/reg.h header file to determine where registers are stored in the exception frame.

The savedefp variable contains the location of the exception frame. (Note that no exception frames are created when you force a system to dump, as described in Section 4.7.) The following example shows an example exception frame:

```
(dbx) print savedefp/33X
ffffffff9618d940:  0000000000000000 fffffc000046f888
ffffffff9618d950:  ffffffff86329ed0 0000000079cd612f
ffffffff9618d960:  000000000000007d 0000000000000001
ffffffff9618d970:  0000000000000000 fffffc000046f4e0
ffffffff9618d980:  0000000000000000 ffffffff9618a2f8
ffffffff9618d990:  0000000140012b20 0000000000000000
ffffffff9618d9a0:  000000014002ee10 0000000000000000
ffffffff9618d9b0:  00000001400075e8 0000000140026240
ffffffff9618d9c0:  ffffffff9618daf0 ffffffff8635af20
ffffffff9618d9d0:  ffffffff9618dac0 00000000000001b0
ffffffff9618d9e0:  fffffc00004941b8 0000000000000000
ffffffff9618d9f0:  0000000000000001 fffffc000028951c
ffffffff9618da00:  0000000000000000 0000000000000fff
ffffffff9618da10:  0000000140026240 0000000000000000
ffffffff9618da20:  0000000000000000 fffffc000047acd0
ffffffff9618da30:  0000000000901402 0000000000001001
ffffffff9618da40:  0000000000002000
```

2.1.8 Extracting the Preserved Message Buffer

The preserved message buffer (pmsgbuf) contains information such as the firmware version, operating system version, pc value, and device configuration. You can use dbx to extract the preserved message buffer from a running system or dump files. For example:

```
(dbx) print *pmsgbuf
struct {
    msg_magic = 405601
    msg_bufx = 1537
    msg_bufr = 1537
    msg_bufc = "Alpha boot: available memory from 0x7c6000 to 0x6000000
Digtal UNIX X4.0-7  (Rev. 5); Sun Jul 03 11:20:36 EST 1995
physical memory = 96.00 megabytes.
available memory = 84.57 megabytes.
using 360 buffers containing 2.81 megabytes of memory
tc0 at nexus
scc0 at tc0 slot 7
asc0 at tc0 slot 6
rz1 at scsi0 target 1 lun 0 (LID=0) (DEC     RZ25     (C) DEC 0700)
rz2 at scsi0 target 2 lun 0 (LID=1) (DEC     RZ25     (C) DEC 0700)
rz3 at scsi0 target 3 lun 0 (LID=2) (DEC     RZ26     (C) DEC T384)
rz4 at scsi0 target 4 lun 0 (LID=3) (DEC     RRD42   (C) DEC  4.5d)
tz5 at scsi0 target 5 lun 0 (DEC     TLZ06     (C)DEC 0374)
scsi1 at tc0 slot 7
fb0 at tc0 slot 8
 1280X1024
ln0: DEC LANCE Module Name: PMAD-BA
ln0 at tc0 slot 7
.
.
.
```

2.1.9 Debugging on SMP Systems

Debugging in an SMP environment can be difficult because an SMP system optimized for performance keeps the minimum of lock debug information.

The Digital UNIX system supports a lock mode to facilitate debugging SMP locking problems. The lock mode is implemented in the `lockmode` boot time system attribute. By default, the `lockmode` attribute is set to a value between 0 and 3, depending upon whether the system is an SMP system and whether the `RT_PREEMPTION_OPT` attribute is set. (This attribute optimizes system performance.)

For debugging purposes, set the `lockmode` attribute to 4. Follow these steps to set the `lockmode` attribute to 4:

1. Create a stanza-formatted file named, for example, `generic.stanza` that appears as follows:

   ```
   generic:
       lockmode=4
   ```

   The contents of this file indicate that you are modifying the `lockmode` attribute of the `generic` subsystem.

2. Add the new definition of `lockmode` to the `/etc/sysconfigtab` database:

   ```
   # sysconfigdb -a -f generic.stanza generic
   ```

3. Reboot your system.

Some of the debugging features provided with `lockmode` set to 4 are as follows:

- Automatic lock hierarchy checking and minimum `spl` checking when any kernel lock is acquired (assuming a `lockinfo` structure exists for the lock class in question). This checking helps you find potential deadlock situations.
- Lock initialization checking.
- Additional debug information maintenance, including information about simple and complex locks.

For simple locks, the system records an array of the last 32 simple locks which were acquired on the system (`slock_debug`). The system creates a `slock_debug` array for each CPU in the system.

For complex locks, the system records the locks owned by each thread in the thread structure (up to eight complex locks).

To get a list of the complex locks a thread is holding use these commands:

```
# dbx -k /vmunix
(dbx) print thread->lock_addr
{
 [0] 0xe4000002a67e0030
 [1] 0xc3e0005b47ff0411
 [2] 0xb67e0030a6130048
 [3] 0xa67e0030d34254e5
 [4] 0x279f0200481e1617
 [5] 0x4ae33738a7730040
 [6] 0x477c0101471c0019
 [7] 0xb453004047210402
}
(dbx) print slock_debug
{
 [0] 0xfffffc000065c580
 [1] 0xfffffc000065c780
}
```

- Lock statistics are recorded to allow you to determine what kind of contention you have on a particular lock. Use the `kdbx lockstats` extension as shown in the following example to display lock statistics:

```
# kdbx /vmunix
(kdbx) lockstats
 Lockstats      li_name           cpu count   tries     misses %misses waitsum     waitmax waitmin  trmax
=========== ===================== === ====== ========== ======= ====== ============ ======= ======= ======
k0x00657d40       inode.i_io_lock   1   1784      74268    1936 2.61      110533     500       6      10
k0x00653400       nfs_daemon_lock   0      1          7       0 0.00           0       0       0       0
k0x00657d80       nfs_daemon_lock   1      1          0       0 0.00           0       0       0       0
k0x00653440                lk_lmf   0      1          0       0 0.00           0       0       0       0
k0x00657dc0                lk_lmf   1      1          2       0 0.00           0       0       0       0
k0x00653480    procfs_global_lock   0      1          3       0 0.00           0       0       0       0
k0x00657e00   procfs_global_lock    1      1          5       0 0.00           0       0       0       0
k0x006534c0  procfs.pr_trace_lock   0     40          0       0 0.00           0       0       0       0
k0x00657e40  procfs.pr_trace_lock   1     40          0       0 0.00           0       0       0
```

2.2 The kdbx Debugger

The kdbx debugger is a crash analysis and kernel debugging tool; it serves as a front end to the dbx debugger. The kdbx debugger is extensible, customizable, and insensitive to changes to offsets and field sizes in structures. The only dependencies on kernel header files are for bit definitions in flag fields.

The kdbx debugger has facilities for interpreting various symbols and kernel data structures. It can format and display these symbols and data structures in the following ways:

- In a predefined form as specified in the source code modules that currently accompany the kdbx debugger
- As defined in user-written source code modules according to a standardized format for the contents of the kdbx modules

All dbx commands (except signals such as Ctrl/P) are available when you use the kdbx debugger. In general, kdbx assumes hexadecimal addresses for commands that perform input and output.

The sections that follow explain using kdbx to debug kernel programs.

2.2.1 Beginning a kdbx Session

Using the kdbx debugger, you can examine the running kernel or dump files created by the savecore utility. In either case, you examine an object file and a core file. For running systems, these files are usually /vmunix and /dev/mem, respectively. The savecore utility saves dump files it creates in the directory specified by the /sbin/init.d/savecore script. By default, the savecore utility saves dump files in the /var/adm/crash directory.

To examine a running system, enter the kdbx command with the following parameters:

```
# kdbx -k /vmunix /dev/mem
```

To examine an object file and core file created by the savecore utility, enter a kdbx command similar to the following:

```
# kdbx -k vmunix.1 vmcore.1
```

The version number (vmunix.*n* and vmcore.*n*) is determined by the value contained in the bounds file, which is located in the default crash directory (/var/adm/crash) or an alternate directory specified by the /sbin/init.d/savecore script.

When you begin a debugging session, kdbx reads and executes the commands in the system initialization file /var/kdbx/system.kdbxrc. The initialization file contains setup commands and alias definitions. (For a list of kdbx aliases, see the kdbx(1) reference page.) You can further customize the kdbx environment by adding commands and aliases to:

- The /var/kdbx/site.kdbxrc file

  This file contains customized commands and alias definitions for a particular system.

- The ~/.kdbxrc file

  This file contains customized commands and alias definitions for a specific user.

- The ./.kdbxrc file

  This file contains customized commands and alias definitions for a specific project. This file must reside in the current working directory when kdbx is invoked.

2.2.2 The kdbx Debugger Commands

The kdbx debugger provides the following commands:

alias [name] [command-string]	Sets or displays aliases. If you omit all arguments, alias displays all aliases. If you specify the variable *name*, alias displays the alias for *name*, if one exists. If you specify *name* and *command-string*, alias establishes *name* as an alias for *command-string*.
context proc | user	Sets context to the user's aliases or the extension's aliases. This command is used only by the extensions.
coredata *start_address* *end_address*	Dumps, in hexadecimal, the contents of the core file starting at *start_address* and ending before *end_address*.

dbx *command-string*	Passes the *command-string* to dbx. Specifying dbx is optional; if kdbx does not recognize a command, it automatically passes that command to dbx. See the dbx(1) reference page for a complete description of dbx commands.
help [-long] [args]	Prints help text.
pr [flags] [extensions] [arguments]	Executes an extension and gives it control of the kdbx session until it quits. You specify the name of the extension in *extension* and pass arguments to it in *arguments*.

–debug	Causes kdbx to display input to and output from the extension on the screen.
–pipe *in_pipe out_pipe*	Used in conjunction with the dbx debugger for debugging extensions. See Chapter 3 for information on using the –pipe flag.
–print_output	Causes the output of the extension to be sent to the invoker of the extension without interpretation as kdbx commands.
–redirect_output	Used by extensions that execute other extensions to redirect the output from the called extensions; otherwise, the user receives the output.
–tty	Causes kdbx to communicate with the subprocess through a terminal line instead of pipes. If you specify the –pipe flag, proc ignores it.

print *string*	Displays *string* on the terminal. If this command is used by an extension, the terminal receives no output.

quit	Exits the kdbx debugger.
source [-x] [file(s)]	Reads and interprets files as kdbx commands in the context of the current aliases. If the you specify the –x flag, the debugger displays commands as they are executed.
unalias *name*	Removes the alias, if any, from *name*.

The kdbx debugger contains many predefined aliases, which are defined in the kdbx startup file /var/kdbx/system.kdbxrc.

2.2.3 Using kdbx Debugger Extensions

In addition to its commands, the kdbx debugger provides extensions. You execute extensions using the kdbx command pr. For example, to execute the arp extension, you enter this command:

```
kdbx> pr arp
```

Some extensions are provided with your Digital UNIX system and reside in the /var/kdbx directory. Aliases for each of these extensions are also provided that let you omit the pr command from an extension command line. Thus, another way to execute the arp extension is to enter the following command:

```
kdbx> arp
```

This command has the same effect as the pr arp command.

You can create your own kdbx extensions as described in Chapter 3.

For extensions that display addresses as part of their output, some use a shorthand notation for the upper 32-bits of an address to keep the output readable. The following table lists the notation for each address type.

Notation	Address Type	Replaces	Example
v	virtual	ffffffff	v0x902416f0
e	virtual	fffffffe	e0x12340000
k	kseg	fffffc00	k0x00487c48
u	user space	00000000	u0x86406200
?	Unrecognized or random type		?0x3782cc33

The sections that follow describe the kdbx extensions that are supplied with your system.

2.2.3.1 Displaying the Address Resolution Protocol Table

The arp extension displays the contents of the address resolution protocol (arp) table. The arp extension has the following form:

arp [–]

If you specify the optional hyphen (–), arp displays the entire arp table; otherwise, it displays those entries that have nonzero values in the iaddr.s_addr and at_flags fields.

For example:

```
(kdbx) arp
       NAME        BUCK SLOT        IPADDR        ETHERADDR MHOLD TIMER FLAGS
================== ==== ====  ============  ============== ===== ===== =====
sys1.zk3.dec.com     11    0  16.140.128.4   170.0.4.0.91.8     0   450     3
sys2.zk3.dec.com     18    0  16.140.128.1     0.0.c.1.8.e8     0   194     3
sys3.zk3.dec.com     31    0  16.140.128.6  8.0.2b.24.23.64     0   539   103
```

2.2.3.2 Performing Commands on Array Elements

The array_action extension performs a command action on each element of an array. This extension allows you to step through any array in the operating system kernel and display specific components or values as described in the list of command flags.

This extension has the following format:

array_action "*type*" *length start_address* [*flags*] *command*

The arguments to the array_action extension are as follows:

"*type*" The type of address of an element in the specified array.

length The number of elements in the specified array.

start_address The address of an array. The address can be specified as a variable name or a number. The more common syntax or notation used to refer to the *start_address* is usually of the form &arrayname[0].

flags

If the you specify the –head flag, the next argument appears as the table header.

If the you specify the –size flag, the next argument is used as the array element size; otherwise, the size is calculated from the element type.

If the you specify the –cond flag, the next argument is used as a filter. It is evaluated by dbx for each array element, and if it evaluates to TRUE, the action is taken on the element. The same substitutions that are applied to the command are applied to the condition.

command

The kdbx or dbx command to perform on each element of the specified array.

Note

The kdbx debugger includes several aliases, such as file_action, that may be easier to use than using the array_action extension directly.

Substitutions similar to printf can be performed on the command for each array element. The possible substitutions are as follows:

Conversion Character	Description
%a	Address of element
%c	Cast of address to pointer to array element
%i	Index of element within the array
%s	Size of element
%t	Type of pointer to element

For example:

```
(kdbx) array_action "struct kernargs *" 11 &kernargs[0] p %c.name
0xfffffc00004737f8 = "askme"
0xfffffc0000473800 = "bufpages"
0xfffffc0000473810 = "nbuf"
0xfffffc0000473818 = "memlimit"
```

```
0xfffffc0000473828 = "pmap_debug"
0xfffffc0000473838 = "syscalltrace"
0xfffffc0000473848 = "boothowto"
0xfffffc0000473858 = "do_virtual_tables"
0xfffffc0000473870 = "netblk"
0xfffffc0000473878 = "zalloc_physical"
0xfffffc0000473888 = "trap_debug"
(kdbx)
```

2.2.3.3 Displaying the Buffer Table

The `buf` extension displays the `buffer` table. This extension has the following format:

buf [*addresses*–free|–all]

If you omit arguments, the debugger displays the buffers on the hash list.

If you specify addresses, the debugger displays the buffers at those addresses. Use the `–free` flag to display buffers on the free list. Use the `–all` flag to display first buffers on the hash list, followed by buffers on the free list.

For example:

```
(kdbx) buf
BUF         MAJ   MIN   BLOCK COUNT  SIZE RESID VNO         FWD         BACK        FLAGS
=========== === =====  ====== ===== ===== ===== =========== =========== =========== ===========
Bufs on hash lists:
v0x904e1b30   8     2   54016  8192  8192     0 v0x902220d0 v0x904f23a8 v0x904e1d20 write cache
v0x904e21f8   8  1025  131722  1024  8192     0 v0x90279800 v0x904e3748 v0x904e22f0 write cache
v0x904e46c8   8  1025  107952  2048  8192     0 v0x90220fa8 v0x904e22f0 v0x904e23e8 read cache
v0x904e9ef0   8  2050  199216  8192  8192     0 v0x90221560 v0x904f2b68 v0x904e66c0 read cache
v0x904df758   8  1025  107968  8192  8192     0 v0x90220fa8 v0x904eac80 v0x904df378 write cache
v0x904eb538   8  2050  223840  8192  8192     0 v0x90221560 v0x904ec990 v0x904eb440 read
v0x904e5930   8  2050  379600  8192  8192     0 v0x90221560 v0x904f3fc0 v0x904ec5b0 read cache
v0x904eae70   8  2050  625392  2048  8192     0 v0x90221560 v0x904df378 v0x904e08c8 write cache
v0x904f3ec8   8  1025   18048  8192  8192     0 v0x90220fa8 v0x904dff18 v0x904e1560 write cache
.
.
.
(kdbx)
```

2.2.3.4 Displaying the Callout Table and Absolute Callout Table

The `callout` extension displays the callout table. This extension has the following format:

callout

For example:

```
(kdbx) callout
Processor:                                    0
Current time (in ticks):              615421360

        FUNCTION                   ARGUMENT    TICKS(delta)
=============================    ============  ============
realitexpire                      k0x008ab220        30772
wakeup                            k0x005d98e0        36541
wakeup                            k0x0187a220       374923
thread_timeout                    k0x010ee950       376286
thread_timeout                    k0x0132f220     40724481
realitexpire                      k0x01069950     80436086
thread_timeout                    k0x01bba950     82582849
```

The abscallout extension displays the absolute callout table. This table contains callout entries with the absolute time in fractions of seconds. This extension has the following format:

abscallout

For example:

```
(kdbx)abscallout
Processor:                              0

        FUNCTION                   ARGUMENT      SECONDS
=============================    ===========  =============
psx4_tod_expire                  k0x01580808  86386.734375
psx4_tod_expire                  k0x01580840  172786.734375
psx4_tod_expire                  k0x01580878  259186.734375
psx4_tod_expire                  k0x015808b0  345586.718750
psx4_tod_expire                  k0x015808e8  431986.718750
psx4_tod_expire                  k0x01580920  518386.718750
psx4_tod_expire                  k0x01580958  604786.750000
psx4_tod_expire                  k0x01580990  691186.750000
psx4_tod_expire                  k0x015809c8  777586.750000
psx4_tod_expire                  k0x01580a00  863986.750000
```

2.2.3.5 Casting Information Stored in a Specific Address

The cast extension forces dbx to display part of memory as the specified type and is equivalent to the following command:

dbx print *((*type*)*address*)

The `cast` extension has the following format:

cast *address type*

For example:

```
(kdbx) cast 0xffffffff903e3828 char
'^@'
```

2.2.3.6 Displaying Machine Configuration

The `config` extension displays the configuration of the machine. This extension has the following format:

config

For example:

```
(kdbx) config
Bus #0 (0xfffffc000048c6a0): Name - "tc"  Connected to - "nexus"
        Config 1 - tcconfl1     Config 2 - tcconfl2
        Controller "scc" (0xfffffc000048c970)
(kdbx)
```

2.2.3.7 Converting the Base of Numbers

The `convert` extension converts numbers from one base to another. This extension has the following format:

convert [–in 8|10|16] [–out 2|8|10|16] [*args*]

The `–in` and `–out` flags specify the input and output bases, respectively. If you omit `–in`, the input base is inferred from the arguments. The arguments can be numbers or variables.

For example:

```
(kdbx) convert -in 16 -out 10 864c2a14
2253138452
(kdbx)
```

2.2.3.8 Displaying CPU Use Statistics

The `cpustat` extension displays statistics about CPU use. Statistics displayed include percentages of time the CPU spends in the following states:

- Running user level code
- Running system level code
- Running at a priority set with the `nice()` function
- Idle
- Waiting (idle with input or output pending)

This extension has the following format:

cpustat [–update *n*] [–cpu *n*]

The `–update` flag specifies that `kdbx` update the output every *n* seconds.

The `–cpu` flag controls the CPU for which `kdbx` displays statistics. By default, `kdbx` displays statistics for all CPUs in the system.

For example:

```
(kdbx) cpustat
 Cpu   User (%)    Nice (%) System (%)  Idle (%)   Wait (%)
===== ========== ========== ========== ========== ==========
    0       0.23       0.00       0.08      99.64       0.05
    1       0.21       0.00       0.06      99.68       0.05
```

2.2.3.9 Disassembling Instructions

The `dis` extension disassembles some number of instructions. This extension has the following format:

dis ***start-address*** [*num-instructions*]

The *num-instructions*, argument specifies the number of instructions to be disassembled. The *start-address* argument specifies the starting address of the instructions. If you omit the *num-instructions* argument, 1 is assumed.

For example:

```
(kdbx) dis 0xffffffff864c2a08 5
  [., 0xffffffff864c2a08]       call_pal        0x20001
  [., 0xffffffff864c2a0c]       call_pal        0x800000
  [., 0xffffffff864c2a10]       ldg     $f18, -13304(r3)
  [., 0xffffffff864c2a14]       bgt     r31, 0xffffffff864c2a14
  [., 0xffffffff864c2a18]       call_pal        0x4573d0
(kdbx)
```

2.2.3.10 Displaying Remote Exported Entries

The `export` extension displays the exported entries that are mounted remotely. This extension has the following format:

export

For example:

```
(kdbx) export
ADDR EXPORT        MAJ  MIN   INUM         GEN  MAP FLAGS PATH
================== === ===== =====  ========== ==== ===== =================
0xffffffff863bfe40  8  4098      2  1308854383   -2     0 /cdrom
0xffffffff863bfdc0  8  2050  67619   736519799   -2     0 /usr/users/user2
0xffffffff863bfe00  8  2050  15263   731712009   -2     0 /usr/staff/user
0xffffffff863bfe80  8  1024   6528   731270099   -2     0 /mnt
```

2.2.3.11 Displaying the File Table

The `file` extension displays the file table. This extension has the following format:

file [*addresses*]

If you omit the arguments, the extension displays file entries with nonzero reference counts; otherwise, it displays the file entries located at the specified addresses.

For example:

```
(kdbx) file
Addr        Type  Ref  Msg Fileops      f_data        Cred Offset Flags
=========== ====  ===  === ======= =========== =========== ====== =====
v0x90406000 file    4    0   vnops v0x90259550 v0x863d5540     68 r w
v0x90406058 file    1    0   vnops v0x9025b5b8 v0x863d5e00   4096 r
v0x904060b0 file    1    0   vnops v0x90233908 v0x863d5d60      0 r
v0x90406108 file    2    0   vnops v0x90233908 v0x863d5d60    602 w
v0x90406160 file    2    0   vnops v0x90228d78 v0x863d5b80    904 r
v0x904061b8 sock    2    0 sockops v0x863b5c08 v0x863d5c20      0 r w
v0x90406210 file    1    0   vnops v0x90239e10 v0x863d5c20   2038 r
v0x90406268 file    1    0   vnops v0x90245140 v0x863d5c20    301 w a
```

```
v0x904062c0 file    3    0   vnops v0x90227880 v0x863d5900     23 r w
v0x90406318 file    2    0   vnops v0x90228b90 v0x863d5c20    856 r
v0x90406370 sock    2    0 sockops v0x863b5a08 v0x863d5c20      0 r w
.
.
.
```

2.2.3.12 Displaying the udb and tcb Tables

The `inpcb` extension displays the `udb` and `tcb` tables. This extension has the following format:

inpcb [–udp] [–tcp] [*addresses*]

If you omit the arguments, `kdbx` displays both tables. If you specify the –`udp` flag or the –`tcp` flag, the debugger displays the corresponding table.

If you specify the *address* argument, the `inpcb` extension ignores the –`udp` and –`tcp` flags and displays entries located at the specified address.

For example:

```
(kdbx) inpcb -tcp
TCP:
   Foreign Host   FPort      Local Host  LPort      Socket         PCB  Options
0.0.0.0              0 0.0.0.0         47621 u0x00000000 u0x00000000
system.dec.com     6000 comput.dec.com   1451 v0x8643f408 v0x863da408
system.dec.com      998 comput.dec.com   1020 v0x8643fc08 v0x863da208
system.dec.com      999 comput.dec.com    514 v0x8643ac08 v0x8643d008
system.dec.com     6000 comput.dec.com   1450 v0x863fba08 v0x863dad08
system.dec.com     1008 comput.dec.com   1021 v0x86431e08 v0x86414708
system.dec.com     1009 comput.dec.com    514 v0x86412808 v0x8643ce08
system.dec.com     6000 comput.dec.com   1449 v0x86436608 v0x86415e08
system.dec.com     6000 comput.dec.com   1448 v0x86431808 v0x863daa08
.
.
.
0.0.0.0              0 0.0.0.0           806 v0x863e3e08 v0x863dbe08
0.0.0.0              0 0.0.0.0           793 v0x863d1808 v0x8635a708
0.0.0.0              0 0.0.0.0             0 v0x86394408 v0x8635b008
0.0.0.0              0 0.0.0.0          1024 v0x86394208 v0x8635b108
0.0.0.0              0 0.0.0.0           111 v0x863d1e08 v0x8635b208
```

2.2.3.13 Performing Commands on Lists

The `list_action` extension performs some command on each element of a linked list. This extension provides the capability to step through any linked list in the operating system kernel and display particular components. This extension has the following format:

list_action "*type*" *next-field end-addr start-addr* [*flags*] *command*

The arguments to the `list_action` extension are as follows:

"*type*"	The type of an element in the specified list.
next-field	The name of the field that points to the next element.
end-addr	The value of the next field that terminates the list. If the list is NULL-terminated, the value of the *end-addr* argument is zero (0). If the list is circular, the value of the *end-addr* argument is equal to the *start-addr* argument.
start_addr	The address of the list. This argument can be a variable name or a number address.
flags	Use the –head *header* flag to display the *header* argument as the table header. Use the –cond *arg* flag to filter input as specified by *arg*. The debugger evaluates the condition for each array element, and if it evaluates to true, the action is taken on the element. The same substitutions that are applied to the command are applied to the condition.
command	The debugger command to perform on each element of the list.

The kdbx debugger includes several aliases, such as procaddr, that might be easier than using the list_action extension directly.

The kdbx debugger applies substitutions in the same style as printf substitutions for each command element. The possible substitutions are as follows:

Conversion Character	Description
%a	Address of an element
%c	Cast of an address to a pointer to a list element
%i	Index of an element within the list
%n	Name of the next field
%t	Type of pointer to an element

For example:

```
(kdbx) list_action "struct proc *" p_nxt 0 allproc p \
%c.task.u_address.uu_comm  %c.p_pid
"list_action" 1382
"dbx" 1380
"kdbx" 1379
"dbx" 1301
"kdbx" 1300
"sh" 1296
"ksh" 1294
"csh" 1288
"rlogind" 1287
⋮
```

2.2.3.14 Displaying the lockstats Structures

The lockstats extension displays the lock statistics contained in the lockstats structures. Statistics are kept for each lock class on each CPU in the system. These structures provide the following information:

- The address of the structure
- The class of lock for which lock statistics are being recorded
- The CPU for which the lock statistics are being recorded
- The number of instances of the lock
- The number of times processes have tried to get the lock
- The number of times processes have tried to get the lock and missed
- The percentage of time processes miss the lock
- The total time processes have spent waiting for the lock
- The maximum amount of time a single process has waited for the lock
- The minimum amount of time a single process has waited for the lock

The lock statistics recorded in the lockstats structures are dynamic.

This extension is available only when the lockmode system attribute is set to 4.

This extension has the following format:

lockstats –class *name* |–cpu *number* |–read |–sum |–total |–update *n*

If you omit all flags, `lockstats` displays statistics for all lock classes on all CPUs. The following describes the flags you can use:

`–class` *name* — Displays the `lockstats` structures for the specified lock class. (Use the `lockinfo` command to display information about the names of lock classes.)

`–cpu` *number* — Displays the `lockstats` structures for the specified CPU.

`–read` — Displays the reads, sleeps attributes, and waitsums or misses.

`–sum` — Displays summary data for all CPUs and all lock types.

`–total` — Displays summary data for all CPUs.

`–update` *n* — Updates the display every *n* seconds.

For example:

```
(kdbx) lockstats
 Lockstats     li_name            cpu count   tries     misses %misses  waitsum     waitmax waitmin  trmax
=========== ==================== === ====== ========== ======= ======= ============ ======= ======= ========
k0x00657d40      inode.i_io_lock   1   1784      74268    1936 2.61       110533     500       6       10
k0x00653400      nfs_daemon_lock   0      1          7       0 0.00            0       0       0        0
k0x00657d80      nfs_daemon_lock   1      1          0       0 0.00            0       0       0        0
k0x00653440               lk_lmf   0      1          0       0 0.00            0       0       0        0
k0x00657dc0               lk_lmf   1      1          2       0 0.00            0       0       0        0
k0x00653480   procfs_global_lock   0      1          3       0 0.00            0       0       0        0
k0x00657e00   procfs_global_lock   1      1          5       0 0.00            0       0       0        0
k0x006534c0 procfs.pr_trace_lock   0     40          0       0 0.00            0       0       0        0
k0x00657e40 procfs.pr_trace_lock   1     40          0       0 0.00            0       0       0        0
.
.
.
```

2.2.3.15 Displaying lockinfo Structures

The `lockinfo` extension displays static lock class information contained in the `lockinfo` structures. Each lock class is recorded in one `lockinfo` structure, which contains the following information:

- The address of the structure
- The index into the array of `lockinfo` structures
- The class of lock for which information is provided
- The number of instances of the lock
- The lock flag, as defined in the `/sys/include/sys/lock.h` header file

This extension is available only when the `lockmode` system attribute is set to 4.

This extension has the following format:

lockinfo [–class *name*]

The `–class` flag allows you to display the `lockinfo` structure for a particular class of locks. If you omit the flag, `lockinfo` displays the `lockinfo` structures for all classes of locks.

For example:

```
(kdbx) lockinfo
    Lockinfo        Index            li_name           li_count  li_flgspl
==================  =====  =========================== ========== =========
xfffffc0000652030      3             cfg_subsys_lock         21    0xd0
0xfffffc0000652040      4             subsys_tbl_lock          1    0xc0
0xfffffc0000652050      5             inode.i_io_lock       4348    0x90
0xfffffc0000652060      6             nfs_daemon_lock          1    0xc0
0xfffffc0000652070      7                      lk_lmf          1    0xc0
0xfffffc0000652080      8          procfs_global_lock          1    0xc0
0xfffffc0000652090      9        procfs.pr_trace_lock         40    0xc0
0xfffffc00006520a0     10     procnode.prc_ioctl_lock          0    0xc0
0xfffffc00006520b0     11                 semidq_lock          1    0xc0
0xfffffc00006520c0     12                  semid_lock         16    0xc0
0xfffffc00006520d0     13                   undo_lock          1    0xc0
0xfffffc00006520e0     14                 msgidq_lock          1    0xc0
0xfffffc00006520f0     15                  msgid_lock         64    0xc0
0xfffffc0000652100     16               pgrphash_lock          1    0xc0
0xfffffc0000652110     17          proc_relation_lock          1    0xc0
0xfffffc0000652120     18                pgrp.pg_lock         20    0xd0
```

2.2.3.16 Displaying the Mount Table

The `mount` extension displays the mount table, and has the following format:

mount [–s] [*address*]

The `–s` flag displays a short form of the table. If you specify one or more addresses, `kdbx` displays the mount entries named by the addresses.

For example:

```
(kdbx) mount
 MOUNT        MAJ    MIN     VNODE        ROOTVP      TYPE         PATH                     FLAGS
===========  =====  =====  ============  ==========  ====  =======================        =====
v0x8196bb30    8       0           NULL  v0x8a75f600  ufs    /
loc
v0x8196a910                 v0x8a62de00  v0x8a684e00  nfs    /share/cia/build/alpha.dsk5      ro
v0x8196aae0                 v0x8a646800  v0x8a625400  nfs    /share/xor/build/agosminor.dsk1  ro
v0x8196acb0                 v0x8a684800  v0x8a649400  nfs    /share/buffer/build/submits.dsk2 ro
v0x8196ae80                 v0x8a67ea00  v0x8a774800  nfs    /share/cia/build/goldos.dsk6     ro
```

```
v0x8196b050                v0x8a67c400  v0x8a767800  nfs    /usr/staff/alpha1/user
v0x8196b220                v0x8a651800  v0x8a781000  nfs    /usr/sde
ro
v0x8196b3f0    8     2050   v0x8a61ca00  v0x8a77fe00  ufs    /usr3
loc
v0x8196b5c0    8        7   v0x8a61c000  v0x8a79c200  ufs    /usr2
loc
v0x8196b790    8        6   v0x8a5c4800  v0x8a760600  ufs    /usr
loc
v0x8196b960    0        0   v0x8a5c5000  NULL         procfs /proc
```

2.2.3.17 Displaying the Namecache Structures

The `namecache` extension displays the namecache structures on the system, and has the following format:

namecache

For example:

```
(kdbx) namecache
 namecache      nc_vp      nc_vpid  nc_nlen   nc_dvp         nc_name
 ===========  =========== =======  ======= ============ =============
 v0x9047b2c0  v0x9021f4f8     24       4   v0x9021e5b8  sbin
 v0x9047b310  v0x9021e988      0      11   v0x9021e7a0  swapdefault
 v0x9047b360  v0x9021e5b8      0       2   v0x9021e7a0  ..
 v0x9047b3b0  v0x9021e7a0    199       3   v0x9021e5b8  dev
 v0x9047b400  v0x9021ed58      0       4   v0x9021eb70  rz1g
 v0x9047b4a0  v0x9021f128      0       4   v0x9021e7a0  init
 v0x9047b4f0  v0x9021f310      0       7   v0x9021e5b8  upgrade
 v0x9047b540  v0x9021fab0     20       3   v0x9021e5b8  etc
 v0x9047b590  v0x9021f6e0      0       7   v0x9021f4f8  inittab
 v0x9047b5e0  v0x9021eb70     28       3   v0x9021e5b8  var
 v0x9047b630  v0x9021f310     34       3   v0x9021e5b8  usr
 v0x9047b6d0  v0x9021fc98      0       7   v0x9021eb70  console
 v0x9047b720  v0x9021fe80      0       2   v0x9021e7a0  sh
 v0x9047b770  v0x90220068      0       3   v0x9021f4f8  nls
 v0x9047b810  v0x90220250      0       8   v0x9021e7a0  bcheckrc
 v0x9047b8b0  v0x90220438      0       4   v0x9021e7a0  fsck
 v0x9047b900  v0x90220620      0       5   v0x9021f4f8  fstab
 v0x9047b950  v0x90220808      0       8   v0x9021e7a0  ufs_fsck
 v0x9047b9a0  v0x902209f0      0       4   v0x9021eb70  rz1a
 v0x9047b9f0  v0x90220bd8      0       5   v0x9021eb70  rrz1a
.
.
.
```

2.2.3.18 Displaying Processes' Open Files

The `ofile` extension displays the open files of processes and has the following format.

ofile [–proc *address*|–pid *pid*|–v]

If you omit arguments, ofile displays the files opened by each process. If you specify –proc *address* or –pid *pid* the extension displays the open files owned by the specified process. The –v flag displays more information about the open files.

For example:

```
(kdbx) ofile -pid 1136 -v
Proc=0xffffffff9041e980   pid= 1136
 ADDR_FILE  f_cnt ADDR_VNODE  V_TYPE V_TAG  USECNT  V_MOUNT     INO#  QSIZE
=========== ===== =========== ====== ====== ====== ========== ====== =====
v0x90408520   27  v0x902c1390  VCHR  VT_UFS    3   v0x863abab8   1103     0
v0x90408520   27  v0x902c1390  VCHR  VT_UFS    3   v0x863abab8   1103     0
v0x90408520   27  v0x902c1390  VCHR  VT_UFS    3   v0x863abab8   1103     0
v0x90408368    1  v0x9026e6b8  VDIR  VT_UFS   18   v0x863ab728  64253   512
```

2.2.3.19 Converting the Contents of Memory to Symbols

The paddr extension converts a range of memory to symbolic references and has the following format:

paddr *address number-of-longwords*

The arguments to the paddr extension are as follows:

address — The starting address.

number-of-longwords — The number of longwords to display.

For example:

```
(kdbx) paddr 0xffffffff90be36d8 20
[., 0xffffffff90be36d8]: [h_kmem_free_memory_:824, 0xfffffc000037f47c] 0x0000000000000000
[., 0xffffffff90be36e8]: [., 0xffffffff8b300d30] [hardclock:394, 0xfffffc00002a7d5c]
[., 0xffffffff90be36f8]: 0x0000000000000000 [., 0xffffffff863828a0]
[., 0xffffffff90be3708]: [setconf:133, 0xfffffc00004949b0] [., 0xffffffff90be39f4]
[., 0xffffffff90be3718]: 0x00000000000004e0 [thread_wakeup_prim:858, 0xfffffc0000328454]
[., 0xffffffff90be3728]: 0x0000000000000001 0xffffffff0000000c
[., 0xffffffff90be3738]: [., 0xffffffff9024e518] [hardclock:394, 0xfffffc00002a7d5c]
[., 0xffffffff90be3748]: 0x00000000004d5ff8 0xffffffffffffffd4
[., 0xffffffff90be3758]: 0x00000000000bc688 [setconf:133, 0xfffffc00004946f0]
[., 0xffffffff90be3768]: [thread_wakeup_prim:901, 0xfffffc00003284d0] 0x000003ff85ef4ca0
```

2.2.3.20 Displaying the Process Control Block for a Thread

The pcb extension displays the process control block for a given thread structure located at *thread_address*. The extension also displays the contents of integer and floating-point registers (if nonzero).

This extension has the following format:

pcb *thread_address*

For example:

```
(kdbx) pcb 0xffffffff863a5bc0
Addr pcb      ksp            usp        pc                   ps
v0x90e8c000   v0x90e8fb88    0x0        0xfffffc00002dc110   0x5
sp                     ptbr       pcb_physaddr
0xffffffff90e8fb88     0x2ad4     0x55aa000
r9   0xffffffff863a5bc0
r10  0xffffffff863867a0
r11  0xffffffff86386790
r13  0x5
```

2.2.3.21 Formatting Command Arguments

The `printf` extension formats one argument at a time to work around the `dbx` debugger's command length limitation. It also supports the `%s` string substitution, which the `dbx` debugger's `printf` command does not. This extension has the following format:

printf *format-string* [*args*]

The arguments to the `printf` extension are as follows:

format-string	A character string combining literal characters with conversion specifications.
args	The arguments for which you want `kdbx` to display values.

For example:

```
(kdbx) printf "allproc = 0x%lx" allproc
allproc = 0xffffffff902356b0
```

2.2.3.22 Displaying the Process Table

The `proc` extension displays the process table. This extension has the following format:

proc [*address*]

If you specify an address, the proc extension displays only the proc structures at that address; otherwise, the extension displays all proc structures.

For example:

```
(kdbx) proc
:
:
Addr        PID   PPID  PGRP  UID   NICE SIGCATCH P_SIG    Event       Flags
=========== ===== ===== ===== ===== ==== ======== ======== =========== ============
v0x8191e210     0     0     0     0    0 00000000 00000000        NULL in sys
v0x8197cd80     1     0     1     0    0 207a7eff 00000000        NULL in pagv exec
v0x8198a210    13     1    13     0    0 00002000 00000000        NULL in pagv
v0x819a8d80   120     1   120     0    0 00086001 00000000        NULL in pagv
v0x819a8210   122     1   122     0    0 00004001 00000000        NULL in pagv
v0x81a14210  5249     1  5267  1138    0 00081000 00000000        NULL in pagv exec
v0x819b6210   131     1   131     0    0 20006003 00000000        NULL in pagv
v0x81a18d80  5266  5267  5267  1138    0 00080000 00000000        NULL in pagv exec
v0x81a2ed80  5267  4938  5267  1138    0 00007efb 00000000        NULL in pagv exec
v0x81a42d80  5268  5266  5267  1138    0 00004007 00000000        NULL in pagv exec
v0x81a18210  5270  5273  5267  1138    0 00000000 00000000        NULL in pagv exec
v0x8198ed80  5273  5266  5267  1138    0 00000000 00000000        NULL in pagv exec
v0x81a0ad80  5276  5279  5276  1138    0 01880003 00000000        NULL
in pagv ctty exec
v0x81a26d80  5278  5249  5278  1138    0 00080002 00000000        NULL
in pagv ctty exec
v0x819f2d80  5279     1  5267  1138    0 00081000 00000000        NULL in pagv exec
v0x81a14d80  5281     1  5267  1138    0 00081000 00000000        NULL in pagv exec
v0x81a3cd80  5287  5281  5287  1138    0 01880003 00000000        NULL
in pagv ctty exec
v0x81a28210  5301  5276  5301  1138    0 00080002 00000000        NULL
in pagv ctty exec
v0x819aad80   195     1   195     0    0 00080628 00000000        NULL in pagv
v0x8197c210  6346     1  6346     0    0 00004006 00000000        NULL in pagv exec
v0x819c4210   204     1     0     0    0 00086efe 00000000        NULL in pagv
:
```

2.2.3.23 Converting an Address to a Procedure name

The procaddr extension converts the specified address to a procedure name. This extension has the following format:

procaddr [*address*]

For example:

```
(kdbx) procaddr callout.c_func
xpt_pool_free
```

2.2.3.24 Displaying Sockets from the File Table

The socket extension displays those files from the file table that are sockets with nonzero reference counts. This extension has the following format:

socket

For example:

```
(kdbx) socket
  Fileaddr   Sockaddr   Type       PCB    Qlen Qlim Scc  Rcc
=========== =========== ===== =========== ==== ==== ===  ====
v0x904061b8 v0x863b5c08 DGRAM v0x8632dc88   0    0   0   0
v0x90406370 v0x863b5a08 DGRAM v0x8632db08   0    0   0   0
v0x90406478 v0x863b5808 DGRAM v0x8632da88   0    0   0   0
v0x904064d0 v0x863b5608 DGRAM v0x8632d688   0    0   0   0
v0x904065d8 v0x863b5408 DGRAM v0x8632dc08   0    0   0   0
v0x90406630 v0x863b5208 DGRAM v0x8632d588   0    0   0   0
v0x904067e8 v0x863b4208 DGRAM v0x8632d608   0    0   0   0
v0x90406840 v0x863b4008 DGRAM v0x8632d788   0    0   0   0
v0x904069a0 v0x8641f008  STRM v0x8632c808   0    0   0   0
v0x90406aa8 v0x863b4c08  STRM v0x8632d508   0    2   0   0
v0x90406bb0 v0x863b4e08  STRM v0x8632da08   0    0   0   0
⋮
```

2.2.3.25 Displaying a Summary of the System Information

The sum extension displays a summary of system information and has the following format:

sum

For example:

```
(kdbx) sum
Hostname : system.dec.com
cpu: DEC3000 - M500     avail: 1
Boot-time:      Tue Nov  3 15:01:37 1992
Time:   Fri Nov  6 09:59:00 1992
Kernel : OSF1 release 1.2 version 1.2 (alpha)
(kdbx)
```

2.2.3.26 Displaying a Summary of Swap Space

The `swap` extension displays a summary of swap space and has the following format:

swap

For example:

```
(kdbx) swap
       Swap device name              Size       In Use       Free
--------------------------------  ----------  ----------  ----------
/dev/rz3b                            131072k      32424k      98648k  Dumpdev
                                      16384p       4053p      12331p
/dev/rz2b                            131072k          8k     131064k
                                      16384p          1p      16383p
--------------------------------  ----------  ----------  ----------
Total swap partitions:    2          262144k      32432k     229712k
                                      32768p       4054p      28714p
(kdbx)
```

2.2.3.27 Displaying the Task Table

The `task` extension displays the task table. This extension has the following format:

task [*proc_address*]

If you specify addresses, the extension displays the task structures named by the argument addresses; otherwise, the debugger displays all tasks.

For example:

```
(kdbx) task
:
:
Task Addr   Ref Threads     Map     Swap_state  Utask Addr  Proc Addr   Pid
=========== === ======= =========== ========== =========== =========== ======
v0x8191e000  17      15 v0x808f7ef0  INSWAPPED v0x8191e3b0 v0x8191e210      0
v0x8197cb70   3       1 v0x808f7760  INSWAPPED v0x8197cf20 v0x8197cd80      1
v0x8198a000   3       1 v0x808f7550  INSWAPPED v0x8198a3b0 v0x8198a210     13
v0x819a8b70   3       1 v0x808f7340  INSWAPPED v0x819a8f20 v0x819a8d80    120
v0x819a8000   3       1 v0x808f7290  INSWAPPED v0x819a83b0 v0x819a8210    122
v0x81a14000   3       1 v0x819f1ad0  INSWAPPED v0x81a143b0 v0x81a14210   5249
v0x819b6000   3       1 v0x808f6fd0  INSWAPPED v0x819b63b0 v0x819b6210    131
v0x81a18b70   3       1 v0x819f1a20  INSWAPPED v0x81a18f20 v0x81a18d80   5266
v0x81a2eb70   3       1 v0x819f1340  INSWAPPED v0x81a2ef20 v0x81a2ed80   5267
v0x81a42b70   3       1 v0x819f1080  INSWAPPED v0x81a42f20 v0x81a42d80   5268
v0x81a18000   3       1 v0x819f1970  INSWAPPED v0x81a183b0 v0x81a18210   5270
v0x8198eb70   3       1 v0x808f74a0  INSWAPPED v0x8198ef20 v0x8198ed80   5273
v0x81a0ab70   3       1 v0x819f1ce0  INSWAPPED v0x81a0af20 v0x81a0ad80   5276
v0x81a26b70   3       1 v0x819f1760  INSWAPPED v0x81a26f20 v0x81a26d80   5278
v0x819f2b70   3       1 v0x819f1e40  INSWAPPED v0x819f2f20 v0x819f2d80   5279
```

```
v0x81a14b70   3       1 v0x819f1b80  INSWAPPED v0x81a14f20 v0x81a14d80   5281
v0x81a3cb70   3       1 v0x819f11e0  INSWAPPED v0x81a3cf20 v0x81a3cd80   5287
v0x81a28000   3       1 v0x819f1550  INSWAPPED v0x81a283b0 v0x81a28210   5301
v0x819aab70   3       1 v0x808f71e0  INSWAPPED v0x819aaf20 v0x819aad80    195
v0x8197c000   3       1 v0x808f76b0  INSWAPPED v0x8197c3b0 v0x8197c210   6346
v0x819c4000   3       1 v0x808f6e70  INSWAPPED v0x819c43b0 v0x819c4210    204
 .
 .
 .
```

2.2.3.28 Displaying Information About Threads

The `thread` extension displays information about threads and has the following format:

thread [*proc_address*]

If you specify addresses, the `thread` extensions displays thread structures named by the addresses; otherwise, information about all threads is displayed.

For example:

```
(kdbx) thread
Thread Addr  Task Addr    Proc Addr    Event        pcb          state
===========  ===========  ===========  ===========  ===========  =====
v0x8644d690  v0x8637e440  v0x9041e830  v0x86420668  v0x90f50000  wait
v0x8644d480  v0x8637e1a0  v0x9041eec0  v0x86421068  v0x90f48000  wait
v0x863a17b0  v0x86380ba0  v0x9041db10  v0x8640e468  v0x90f30000  wait
v0x863a19c0  v0x86380e40  v0x9041d9c0  v0x8641f268  v0x90f2c000  wait
v0x8644dcc0  v0x8637ec20  v0x9041e6e0  v0x8641fc00  v0x90f38000  wait
v0x863a0520  v0x8637f400  v0x9041ed70  v0x8640ea00  v0x90f3c000  wait
v0x863a0310  v0x8637f160  v0x9041e980  u0x00000000  v0x90f44000  run
v0x863a2410  v0x863818c0  v0x9041dc60  v0x8640f268  v0x90f18000  wait
v0x863a15a0  v0x86380900  v0x9041d480  v0x8641ec00  v0x90f24000  wait
 .
 .
 .
```

2.2.3.29 Displaying a Stack Trace of Threads

The `trace` extension displays the stack of one or more threads. This extension has the following format:

trace [*thread_address...* –k|–u|–a]

If you omit arguments, `trace` displays the stack trace of all threads. If you specify a list of thread addresses, the debugger displays the stack trace of the specified threads. The following table explains the `trace` flags:

–a   Displays the stack trace of the active thread on each CPU

–k Displays the stack trace of all kernel threads

–u Displays the stack trace of all user threads

For example:

```
(kdbx) trace
*** stack trace of thread 0xffffffff819af590  pid=0 ***
>  0 thread_run(new_thread = 0xffffffff819af928)
["../../../../src/kernel/kern/sched_prim.c":1637, 0xfffffc00002f9368]
   1 idle_thread() ["../../../../src/kernel/kern/sched_prim.c":2717,
0xfffffc00002fa32c]
*** stack trace of thread 0xffffffff819af1f8  pid=0 ***
>  0 thread_block() ["../../../../src/kernel/kern/sched_prim.c":1455,
0xfffffc00002f9084]
   1 softclock_main() ["../../../../src/kernel/bsd/kern_clock.c":810,
0xfffffc000023a6d4]
.
.
.
*** stack trace of thread 0xffffffff819fc398  pid=0 ***
>  0 thread_block() ["../../../../src/kernel/kern/sched_prim.c":1471,
0xfffffc00002f9118]
   1 vm_pageout_loop() ["../../../../src/kernel/vm/vm_pagelru.c":375,
0xfffffc0000395664]
   2 vm_pageout() ["../../../../src/kernel/vm/vm_pagelru.c":834,
0xfffffc00003961e0]
.
.
.
*** stack trace of thread 0xffffffff819fce60  pid=2 ***
>  0 thread_block() ["../../../../src/kernel/kern/sched_prim.c":1471,
0xfffffc00002f9118]
   1 msg_dequeue(message_queue = 0xffffffff819a5970, max_size = 8192,
option = 0, tout = 0, kmsgptr = 0xffffffff916e3980)
["../../../../src/kernel/kern/ipc_basics.c":884, 0xfffffc00002e8b54]
   2 msg_receive_trap(header = 0xfffffc00005bc150, option = 0, size =
8192, name = 0, tout = 0)
["../../../../src/kernel/kern/ipc_basics.c":1245, 0xfffffc00002e92a4]
   3 msg_receive(header = 0xfffffc00005be150, option = 6186352, tout =
0) ["../../../../src/kernel/kern/ipc_basics.c":1107, 0xfffffc00002e904c]
   4 ux_handler() ["../../../../src/kernel/builtin/ux_exception.c":221,
0xfffffc000027269c]
*** stack trace of thread 0xffffffff81a10730  pid=13 ***
>  0 thread_block() ["../../../../src/kernel/kern/sched_prim.c":1471,
0xfffffc00002f9118]
   1 mpsleep(chan = 0xffffffff819f3270 =

"H4\237\201\377\377\377\377^X0\237\201\377\377\377\377^ ^YR", pri =
296, wmesg = 0xfffffc000042f5e0 =

"\200B\260\300B\244KA\340\3038F]\244\377, timo = 0,
lockp = (nil), flags = 0)
["../../../../src/kernel/bsd/kern_synch.c":341, 0xfffffc0000250250]
   2 sigsuspend(p = 0xffffffff81a04278, args = 0xffffffff9170b8a8,
retval = 0xffffffff9170b898)
.
.
.
```

2.2.3.30 Displaying a u Structure

The u extension displays a u structure. This extension has the following format:

u [*proc-addr*]

If you omit arguments, the extension displays the u structure of the currently running process.

For example:

```
(kdbx) u ffffffff9027ff38
procp   0x9027ff38
ar0     0x90c85ef8
comm     cfgmgr
args      g    B*  ü
u_ofile_of: 0x86344e30 u_pofile_of: 0x86345030
 0 0xffffffff902322d0
 1 0xffffffff90232278
 2 0xffffffff90232278
 3 0xffffffff90232328
 4 0xffffffff90232380  Auto-close
 5 0xffffffff902324e0
sizes   29 45 2 (clicks)
u_outime        0
sigs
           40    40    40    40    40    40    40    40
           40    40    40    40    40    40    40    40
           40    40    40    40    40    40    40    40
           40    40    40    40    40    40    40    40
sigmask
            0 fffefeff fffefeff fffefeff     0     0     0     0
            0     0     0     0     0 fffefeff     0 fffefeff
            0     0     0     0     0     0     0     0
            0     0     0     0     0     0     0     0
sigonstack            0
oldmask            2000
sigstack              0               0
cdir rdir        901885b8         0
timers
start                      0    723497702
acflag          193248
(kdbx)
```

2.2.3.31 Displaying References to the ucred Structure

The `ucred` extension displays all instances of references to `ucred` structures. This extension has the following format:

ucred [–proc|–uthread|–file|–buf|–ref *addr*|–check *addr*|checkall]

If you omit all flags, `ucred` displays all references to `ucred` structures. The following describes the flags you can specify:

`-proc` Displays all `ucreds` referenced by the `proc` structures

`-uthread` Displays all `ucreds` referenced by the `uthread` structures

`-file` Displays all `ucreds` referenced by the file structures

`-buf` Displays all `ucreds` referenced by the `buf` structures

`-ref` *address* Displays all references to a given `ucred`

`-check` *address* Checks the reference count of a particular `ucred`

`-checkall` Checks the reference count of all `ucreds`, with mismatches marked by an asterisk (`*`)

For example:

```
(kdbx) ucred
    ADDR OF UCRED      ADDR OF Ref      Ref Type cr_ref cr_uid cr_gid cr_ruid
===================  ================== ======== ====== ====== ====== =======
0xffffffff863d4960   0xffffffff90420f90    proc      3       0      1      0
0xffffffff8651fb80   0xffffffff9041e050    proc     18       0      1      0
0xffffffff86525c20   0xffffffff90420270    proc      2       0      1      0
0xffffffff86457ea0   0xffffffff90421380    proc      4    1139     15   1139
0xffffffff86457ea0   0xffffffff9041f6a0    proc      4    1139     15   1139
0xffffffff8651b5e0   0xffffffff9041f010    proc      2       0      1      0
0xffffffff8651efa0   0xffffffff9041e1a0    proc      2    1138     10   1138
.
.
0xffffffff863d4960   0xffffffff90fb82e0  uthread     3       0      1      0
0xffffffff8651fb80   0xffffffff90fbc2e0  uthread    18       0      1      0
0xffffffff86525c20   0xffffffff90fb02e0  uthread     2       0      1      0
0xffffffff86457ea0   0xffffffff90f882e0  uthread     4    1139     15   1139
0xffffffff86457ea0   0xffffffff90f902e0  uthread     4    1139     15   1139
0xffffffff8651b5e0   0xffffffff90fc02e0  uthread     2       0      1      0
0xffffffff8651efa0   0xffffffff90fac2e0  uthread     2    1138     10   1138
.
.
0xffffffff863d5c20   0xffffffff90406790     file    16       0      0      0
```

```
0xffffffff863d5b80   0xffffffff904067e8     file     7       0      0      0
0xffffffff863d5c20   0xffffffff90406840     file    16       0      0      0
0xffffffff863d5b80   0xffffffff90406898     file     7       0      0      0
0xffffffff86456000   0xffffffff904068f0     file    15    1139     15   1139
0xffffffff863d5c20   0xffffffff90406948     file    16       0      0      0
:
(kdbx) ucred -ref 0xffffffff863d5a40
    ADDR OF UCRED      ADDR OF Ref      Ref Type cr_ref cr_uid cr_gid cr_ruid
=================== ================== ======== ====== ====== ====== =======
0xffffffff863d5a40   0xffffffff9041c0d0    proc      4       0      0      0
0xffffffff863d5a40   0xffffffff90ebc2e0  uthread     4       0      0      0
0xffffffff863d5a40   0xffffffff90406f78     file     4       0      0      0
0xffffffff863d5a40   0xffffffff90408730     file     4       0      0      0
(kdbx) ucred -check 0xffffffff863d5a40
   ADDR OF UCRED     cr_ref   Found
==================  ======  =======
0xffffffff863d5a40      4      4
```

2.2.3.32 Removing Aliases

The `unaliasall` extension removes all aliases, including the predefined aliases. This extension has the following format:

unaliasall

For example:

```
(kdbx) unaliasall
```

2.2.3.33 Displaying the vnode Table

The `vnode` extension displays the `vnode` table and has the following format:

vnode [–free|–all|–ufs|–nfs|–cdfs|–advfs|–fs *address*|–u *uid*|–g *gid*|–v]

If you omit flags, `vnode` displays ACTIVE entries in the `vnode` table. (ACTIVE means that `usecount` is nonzero.) The following describes the flags you can specify:

`-free` Displays INACTIVE entries in the `vnode` table

`-all` Prints ALL (both ACTIVE and INACTIVE) entries in the `vnode` table

`-ufs` Displays all UFS entries in the `vnode` table

`-nfs` Displays all NFS entries in the `vnode` table

`-cdfs`	Displays all CDFS entries in the `vnode` table
`-advfs`	Displays all ADVFS entries in the `vnode` table
`-fs` *address*	Displays the `vnode` entries of a mounted file system
`-u` *uid*	Displays `vnode` entries of a particular user
`-g` *gid*	Displays `vnode` entries of a particular group
`-v`	Displays related `inode`, `rnode`, or `cdnode` information (used with `-ufs`, `-nfs`, or `-cdfs` only)

For example:

```
(kdbx) vnode
ADDR_VNODE  V_TYPE  V_TAG USECNT V_MOUNT
=========== ====== ====== ====== ===========
v0x9021e000  VBLK  VT_NON     1  k0x00467ee8
v0x9021e1e8  VBLK  VT_NON    83  v0x863abab8
v0x9021e3d0  VBLK  VT_NON     1  k0x00467ee8
v0x9021e5b8  VDIR  VT_UFS    34  v0x863abab8
v0x9021e7a0  VDIR  VT_UFS     1  v0x863abab8
v0x9021ed58  VBLK  VT_UFS     1  v0x863abab8
v0x9021ef40  VBLK  VT_NON     1  k0x00467ee8
v0x9021f128  VREG  VT_UFS     3  v0x863abab8
v0x9021f310  VDIR  VT_UFS     1  v0x863abab8
v0x9021f8c8  VREG  VT_UFS     1  v0x863abab8
v0x9021fe80  VREG  VT_UFS     1  v0x863abab8
v0x902209f0  VDIR  VT_UFS     1  v0x863abab8
v0x90220fa8  VBLK  VT_UFS     9  v0x863abab8
v0x90221190  VBLK  VT_NON     1  k0x00467ee8
v0x90221560  VREG  VT_UFS     1  v0x863abab8
v0x90221748  VBLK  VT_UFS  3153  v0x863abab8
.
.
.
(kdbx) vnode -nfs -v
ADDR_VNODE  V_TYPE  V_TAG USECNT  V_MOUNT     FILEID  MODE  UID  GID  QSIZE
=========== ====== ====== ====== ===========  ====== ====== ==== ==== ======
v0x90246820  VDIR  VT_NFS     1  v0x863ab560 205732  40751  1138   23   2048
v0x902471a8  VDIR  VT_NFS     1  v0x863ab398 378880  40755  1138   10   5120
v0x90247578  VDIR  VT_NFS     1  v0x863ab1d0      2  40755     0    0   1024
v0x90247948  VDIR  VT_NFS     1  v0x863ab008 116736  40755  1114    0    512
v0x9026d1c0  VDIR  VT_NFS     1  v0x863ab1d0  14347  40755     0   10    512
v0x9026e8a0  VDIR  VT_NFS     1  v0x863aae40      2  40755     0   10    512
v0x9026ea88  VDIR  VT_NFS     1  v0x863ab1d0  36874  40755     0   10    512
v0x90272788  VDIR  VT_NFS     1  v0x863ab1d0  67594  40755     0   10    512
v0x902fd080  VREG  VT_NFS     1  v0x863ab1d0  49368 100755  8887  177 455168
v0x902ff888  VREG  VT_NFS     1  v0x863ab1d0  49289 100755  8887  177 538200
v0x90326410  VREG  VT_NFS     1  v0x863aae40 294959 100755     3    4 196608
.
.
.
(kdbx) vnode -ufs -v
ADDR_VNODE  V_TYPE  V_TAG USECNT  V_MOUNT     INODE#  MODE  UID  GID  QSIZE
=========== ====== ====== ====== ===========  ====== ====== ==== ==== ======
v0x9021e5b8  VDIR  VT_UFS    34  v0x863abab8      2  40755     0    0   1024
v0x9021e7a0  VDIR  VT_UFS     1  v0x863abab8   1088  40755     0    0   2560
v0x9021ed58  VBLK  VT_UFS     1  v0x863abab8   1175  60600     0    0      0
```

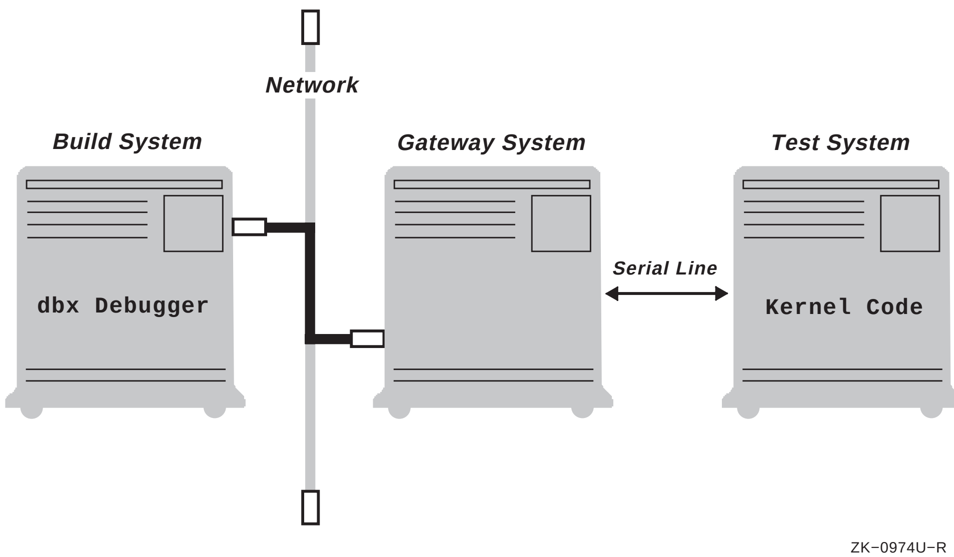
```
v0x9021f128  VREG  VT_UFS     3  v0x863abab8   7637 100755     3    4 147456
v0x9021f310  VDIR  VT_UFS     1  v0x863abab8   8704  40755     3    4    512
v0x9021f8c8  VREG  VT_UFS     1  v0x863abab8   7638 100755     3    4  90112
v0x9021fe80  VREG  VT_UFS     1  v0x863abab8   7617 100755     3    4 196608
v0x902209f0  VDIR  VT_UFS     1  v0x863abab8   9792  41777     0   10    512
v0x90220fa8  VBLK  VT_UFS     9  v0x863abab8   1165  60600     0    0      0
v0x90221560  VREG  VT_UFS     1  v0x863abab8   7635 100755     3    4 245760
v0x90221748  VBLK  VT_UFS  3151  v0x863abab8   1184  60600     0    0      0
 .
 .
 .
```

2.3 The kdebug Debugger

The kdebug debugger allows you to debug running kernel programs. You can start and stop kernel execution, examine variable and register values, and perform other debugging tasks, just as you would when debugging user space programs.

The ability to debug a running kernel is provided through remote debugging. The kernel code you are debugging runs on a test system. The dbx debugger runs on a remote build system. The debugger communicates with the kernel code you are debugging over a serial communication line or through a gateway system. You use a gateway system when you cannot physically connect the test and build systems. Figure 2–1 shows the connections needed when you use a gateway system.

Figure 2–1: Using a Gateway System During Remote Debugging



As shown in Figure 2–1, when you use a gateway system, the build system is connected to it using a network line. The gateway system is connected to the test system using a serial communication line.

Prior to running the kdebug debugger, the test, build, and gateway systems must meet the following requirements:

- The test system must be running Digital UNIX Version 2.0 or higher, must have the Kernel Debugging Tools subset loaded, and must have the Kernel Breakpoint Debugger kernel option configured.
- The build system must be running Digital UNIX Version 2.0 or higher and must have the Kernel Debugging Tools subset loaded. Also, this system must contain a copy of the kernel code you are testing and, preferably, the source used to build that kernel code.
- The gateway system must be running Digital UNIX Version 2.0 or higher and must have the Kernel Debugging Tools subset loaded.

To use the kdebug debugger, you must set up your build, gateway, and test systems as described in Section 2.3.1. Once you complete the setup, you invoke dbx as described in Section 2.3.2 and enter commands as you normally would. Refer to Section 2.3.3 if you have problems with the setup of your remote kdebug debugging session.

2.3.1 Getting Ready to Use the kdebug Debugger

To use the kdebug debugger, you must do the following:

1. Attach the test system and the build (or gateway) system.

   To attach the serial line between the test and build (or gateway) systems, locate the serial line used for kernel debugging. In general, the correct serial line is either /dev/tty00 or /dev/tty01. For example, if you have a DEC 3000 family workstation, kdebug debugger input and output is always to the RS232C port on the back of the system. By default, this port is identified as /dev/tty00 at installation time.

   If your system is an AlphaStation or AlphaServer system with an ace console serial interface, the system uses one of two serial ports for kdebug input and output. By default, these systems use the COMM1 serial port (identified as /dev/tty00) when operating as a build or gateway system. These systems use the COMM2 serial port (identified as /dev/tty01) when operating as the test system.

   To make it easier to connect the build or gateway system and the test system for kernel debugging, you can modify your system setup. You can change the system setup so that the COMM2 serial port is always used for kernel debugging whether the system is operating as a build system, a gateway system, or a test system.

   To make COMM2 the serial port used for kernel debugging on AlphaStations and AlphaServers, modify your /etc/remote file. On these systems, the default kdebug debugger definition in the /etc/remote file appears as follows:

   ```
   kdebug:dv=/dev/tty00:br#9600:pa=none:
   ```

   Modify this definition so that the device is /dev/tty01 (COMM2), as follows:

   ```
   kdebug:dv=/dev/tty01/br#9600:pa=none:
   ```

2. On the build system, install the Product Authorization Key (PAK) for the Developer's kit (OSF-DEV), if it is not already installed. For the gateway and tests systems, the OSF-BASE license PAK is all that is needed. For information about installing PAKs, see the *Software License Management* guide.

3. On the build system, modify the setting of the $kdebug_host, $kdebug_line, or $kdebug_dbgtty as needed.

   The $kdebug_host variable is the name of the gateway system. By default, $kdebug_host is set to localhost, assuming no gateway system is being used.

   The $kdebug_line variable selects the serial line definition to use in the /etc/remote file of the build system (or the gateway system, if one is being used). By default, $kdebug_line is set to kdebug.

   The $kdebug_dbgtty variable sets the terminal on the gateway system to display the communication between the build and test systems, which is useful in debugging your setup. To determine the terminal name to supply to the $kdebug_dbgtty variable, enter the tty command in the correct window on the gateway system. By default, $kdebug_dbgtty is null.

   For example, the following $HOME/.dbxinit file sets the $kdebug_host variable to a system named gatewy:

   ```
   set $kdebug_host="gatewy"
   ```

4. Recompile kernel files, if necessary.

   By default, the kernel is compiled with only partial debugging information. Occasionally, this partial information causes kdebug to display erroneous arguments or mismatched source lines. To correct this, recompile selected source files on the test system specifying the CDEBUGOPTS=–g argument.

5. Make a backup copy of the kernel running on the test system so that you can restore that kernel after testing:

   ```
   # mv /vmunix /vmunix.save
   ```

6. Copy the kernel to be tested to /vmunix on the test system and reboot the system:

   ```
   # cp vmunix.test /vmunix
   # shutdown -r now
   ```

7. If you are debugging on an SMP system, set the lockmode system attribute to 4 on the test system, as follows:

   a. Create a stanza-formatted file named, for example generic.stanza, that appears as follows:

   ```
   generic:
         lockmode = 4
   ```

   This file indicates that you are modifying the lockmode attribute in the generic subsystem.

   b. Use the sysconfigdb command to add the contents of the file to the /etc/sysconfigtab database:

   ```
   # sysconfigdb -a -f generic.stanza generic
   ```

   c. Reboot your system.

   Setting this system attribute makes debugging on an SMP system easier. For information about the advantages provided see Section 2.1.9.

8. Set the OPTIONS KDEBUG configuration file option in your test kernel. To set this option, run the doconfig command without flags, as shown:

   ```
   # doconfig
   ```

   Choose KERNEL BREAKPOINT DEBUGGING from the kernel options menu when it is displayed by doconfig. Once doconfig finishes building a new kernel, copy that kernel to the /vmunix file and reboot your system. For more information about using the kernel options menu to modify the kernel, see the *System Administration* guide.

2.3.2 Invoking the kdebug Debugger

You invoke the kdebug debugger as follows:

1. Invoke the dbx debugger on the build system, supplying the pathname of the test kernel. Set a breakpoint and start running dbx as follows:

   ```
   # dbx -remote vmunix
   dbx version 3.12.1
   Type 'help' for help.
   main: 602  p = &proc[0];
   ```

```
(dbx) stop in main
[2] stop in main
(dbx) run
```

Note that you can set a breakpoint anytime after the execution of the `kdebug_bootstrap()` routine. Setting a breakpoint prior to the execution of this routine can result in unpredictable behavior.

You can use all valid `dbx` flags with the `-remote` flag and define entries in your `$HOME/.dbxinit` file as usual. For example, suppose you start the `dbx` session in a directory other than the one that contains the source and object files used to build the `vmunix` kernel you are running on the test system. In this case, use the `-I` command flag or the `use` command in your `$HOME/.dbxinit` file to point `dbx` to the appropriate source and object files. For more information, see `dbx`(1) and the *Programmer's Guide*.

2. Halt the test system and, at the console prompt (three right angle brackets), set the `boot_osflags` console variable to contain the `k` option, and then boot the system. For example:

```
>>> set boot_osflags "k"
>>> boot
```

Once you boot the kernel, it begins executing. The `dbx` debugger will halt execution at the breakpoint you specified, and you can begin issuing `dbx` debugging commands. See Section 2.1, the `dbx`(1) reference page, or the *Programmer's Guide* for information on `dbx` debugging commands.

If you are unable to bring your test kernel up to a fully operational mode, you can reboot the halted system running the generic kernel, as follows:

```
>>> set boot_osflags "S"
>>> set boot_file "/genvmunix"
>>> boot
```

Once the system is running, you can run the `bcheckrc` script manually to check and mount your local file systems. Then, copy the appropriate kernel to the `root` (/) directory.

When you are ready to resume debugging, copy the test kernel to /vmunix and reset the console variables and boot the system, as follows:

```
>>> set boot_osflags "k"
>>> set boot_file "/vmunix"
>>> boot
```

When you have completed your debugging session, reset the console variables on the test system to their normal values, as follows:

```
>>> set boot_osflags "A"
>>> set boot_file "/vmunix"
>>> set auto_action boot
```

You might also need to replace the test kernel with a more reliable kernel. For example, you should have saved a copy of the vmunix file that is normally used to run the test system. You can copy that file to /vmunix and shutdown and reboot the system:

```
# mv /vmunix.save /vmunix
# shutdown -r now
```

2.3.3 Diagnosing kdebug Setup Problems

If you have completed the kdebug setup as described in Section 2.3.2 and it fails to work, refer to the following list for help in diagnosing and fixing the setup problem:

- Determine whether the serial line is attached properly and then use the tip command to test the connection.

  Once you determine that the serial line is attached properly, log on to the build system (or the gateway system if one is being used) as root and enter the following command:

  ```
  # tip kdebug
  ```

  If the command does not return the message connected, another process, such as a print daemon, might be using the serial line port that you have dedicated to the kdebug debugger. To remedy this condition, do the following:

– Check the /etc/inittab file to see if any processes are using that line. If so, disable these lines until you finish with the kdebug session. See the inittab(4) reference page for information on disabling lines.

– Examine your /etc/remote file to determine which serial line is associated with the kdebug label. Then, use the ps command to see if any processes are using the line. For example, if you are using the /dev/tty00 serial port for your kdebug session, check for other processes using the serial line with the following command:

```
# ps agxt00
```

If a process is using tty00, either kill that process or modify the kdebug label so that a different serial line is used.

If the serial line specified in your /etc/remote file is used as the system's serial console, do not kill the process. In this case, use another serial line for the kdebug debugger.

– Determine whether any unused kdebugd gateway daemons are running with the following command:

```
# ps agx | grep kdebugd
```

After ensuring the daemons are unused, kill the daemon processes.

- If the test system boots to single user or beyond, then kdebug has not been configured into the kernel as specified in Section 2.3.1. Ensure that the boot_osflags console environment variable specifies the k flag and try booting the system again:

```
>>> set boot_osflags k
>>> boot
```

- Be sure you defined the dbx variables in your $HOME/.dbxinit file correctly.

  Determine which terminal line you ran tip from by issuing the /usr/bin/tty command. For example:

```
# /usr/bin/tty
/dev/ttyp2
```

This example shows that you are using terminal /dev/ttyp2. Edit your $HOME/.dbxinit file on the build system as follows:

– Set the $kdebug_dbgtty variable to /dev/ttyp2 as follows:

```
set $kdebug_dbgtty="/dev/ttyp2"
```

– Set the $kdebug_host variable to the host name of the system from which you entered the tip command. For example, if the host name is MYSYS, the entry in the $HOME/.dbxinit file will be as follows:

```
set $kdebug_host="mysys"
```

– Remove any settings of the $kdebug_line variable as follows:

```
set $kdebug_line=
```

- Start dbx on the build system. You should see informational messages on the terminal line /dev/ttyp2 that kdebug is starting.
- If you are using a gateway system, ensure that the inetd daemon is running on the gateway system. Also, check the TCP/IP connection between the build and gateway systems using one of the following commands: rlogin, rsh, or rcp.

2.3.4 Notes on Using the kdebug Debugger

The following list contains information that can help you use the kdebug debugger effectively:

- Breakpoint behavior on SMP systems

  If you set breakpoints in code that is executed on an SMP system, the breakpoints are handled serially. When a breakpoint is encountered on a particular CPU, the state of all the other processors in the system is saved and those processors spin. This behavior is similar to how execution stops when a simple lock is obtained on a particular CPU.

  Processing resumes on all processors when the breakpoint is dismissed; for example, when you enter a step or cont command to the debugger.

- Reading instructions from disk

  By default, the dbx debugger reads instructions from the remote kernel's memory. Reading instructions from memory allows the debugger to help you examine self-modifying code, such as spl routines.

You can force the debugger to look at instructions in the on-disk copy of the kernel by adding the following line to your `$HOME/.dbxinit` file:

```
set $readtextfile = 1
```

Setting the `$readtextfile` variable might improve the speed of the debugger while it is reading instructions.

Be aware that the instructions the debugger reads from the on-disk copy of the kernel might be made obsolete by self-modifying code. The on-disk copy of the kernel does not contain any modifications made to the code as it is running. Obsolete instructions that the debugger reads from the on-disk copy can cause the kernel to fail in an unpredictable way.

2.4 The crashdc Utility

The `crashdc` utility collects critical data from operating system crash dump files or from a running kernel. You can use the data it collects to analyze the cause of a system crash. The `crashdc` utility uses existing system tools and utilities to extract information from crash dumps. The information garnered from crash dump files or from the running kernel includes the hardware and software configuration, current processes, the panic string (if any), and swap information.

The `crashdc` utility is invoked each time the system is booted. If it finds a current crash dump, `crashdc` creates a data collection file with the same numerical file name extension as the crash dump (see Section 4.5 for information about crash dump names).

You can also invoke `crashdc` manually. The syntax of the command for invoking the data collection script is as follows:

/bin/crashdc vmunix.*n* /vmcore.*n*

See Appendix A for an example of the output from the `crashdc` command.

3

Writing Extensions to the kdbx Debugger

To assist in debugging kernel code, you can write an extension to the kdbx debugger. Extensions interact with kdbx and enable you to examine kernel data relevant to debugging the source program. This chapter provides the following:

- A list of considerations before you begin writing extensions (Section 3.1)
- A description of the kdbx library routines that you can use to write extensions (Section 3.2)
- Examples of kdbx extensions (Section 3.3)
- Instructions for compiling extensions (Section 3.4)
- Information to help you debug your kdbx extensions (Section 3.5)

The Digital UNIX Kernel Debugging Tools subset must be installed on your system before you can create custom extensions to the kdbx debugger. This subset contains header files and libraries needed for building kdbx extensions. See Section 3.1 for more information.

3.1 Basic Considerations for Writing Extensions

Before writing an extension, consider the following:

- The information that is needed

  Identify the kernel variables and symbols that you need to examine.

- The means for displaying the information

  Display the information so that anyone who needs to use it can read and understand it.

- The need to provide useful error checking

  As with any good program, it is important to provide informational error messages in the extension.

Before you write an extension, become familiar with the library routines in the libkdbx.a library. These library routines provide convenient methods of extracting and displaying kernel data. The routines are declared in the /usr/include/kdbx.h header file and described in Section 3.2.

You should also study the extensions that are provided on your system in the /var/kdbx directory. These extensions and the example extensions discussed in Section 3.3 can help you understand what is involved in writing an extension and provide good examples of using the kdbx library functions.

3.2 Standard kdbx Library Functions

The kdbx debugger provides a number of library functions that are used by the resident extensions. You can use these functions, which are declared in the ./usr/include/kdbx.h header file, to develop customized extensions for your application. To use the functions, you must include the ./usr/include/kdbx.h header file in your extension.

The sections that follow describe the special data types defined for use in kdbx extensions and the library routines you use in extensions. The library routine descriptions show the routine syntax and describe the routine arguments. Examples included in the descriptions show significant lines in boldface type.

3.2.1 Special kdbx Extension Data Types

The routines described in this section use the following special data types: StatusType, Status, FieldRec, and DataStruct. The uses of these data types are as follows:

- The StatusType data type is used to declare the status type and can take on any one of the following values:
  - OK, which indicates that no error occurred
  - Comm, which indicates a communication error
  - Local, which indicates other types of errors

  The following is the type definition for the StatusType data type:

  ```
  typedef enum { OK, Comm, Local } StatusType;
  ```

- The Status data type is returned by some library routines to inform the caller of the status of the call. Library routines using this data type fill in the type field with the call status from StatusType. Upon return, callers check the type field, and if it is not set to OK, they can pass the Status structure to the print_status routine to generate a detailed error message.

  The following is the type definition for the Status data type:

```
typedef struct {
  StatusType type;
  union {
    int comm;
    int local;
  } u;
} Status;
```

  The values in comm and local provide the error code interpreted by print_status.

- The FieldRec data type, which is used to declare a field of interest in a data structure.

  The following is the type definition for the FieldRec data type:

```
typedef struct {
  char *name;
  int type;
  caddr_t data;
  char *error;
} FieldRec;
```

  The char **name* declaration is the name of the field in question. The int *type* declaration is the type of the field, for example, NUMBER, STRUCTURE, POINTER. The caddr_t *data* and char **error* declarations are initially set to NULL. The read_field_vals function fills in these values.

- The DataStruct, data type, which is used to declare data structures with opaque data types.

  The following is the type definition for the DataStruct data type:

```
typedef long DataStruct;
```

3.2.2 Converting an Address to a Procedure Name

The `addr_to_proc` function returns the name of the procedure that begins the address you pass to the function. If the address is not the beginning of a procedure, then a string representation of the address is returned. The return value is dynamically allocated by `malloc` and should be freed by the extension when it is no longer needed.

This function has the following syntax:

char *addr_to_proc(long *addr*);

Argument	Input/Output	Description
addr	Input	Specifies the address that you want converted to a procedure name

For example:

```
conf1 = addr_to_proc((long) bus_fields[3].data);
conf2 = addr_to_proc((long) bus_fields[4].data);
sprintf(buf, "Config 1 - %sConfig 2 - %s", conf1, conf2);
free(conf1);
free(conf2);
```

3.2.3 Getting a Representation of an Array Element

The `array_element` function returns a representation of one element of an array. The function returns non-NULL if it succeeds or NULL if an error occurs. When the value of error is non-NULL, the *error* argument is set to point to the error message. This function has the following syntax:

DataStruct array_element(DataStruct *sym* , int *i* , char ** *error*);

Argument	Input/Output	Description
sym	Input	Names the array
i	Input	Specifies the index of the element
error	Output	Returns a pointer to an error message, if the return value is NULL

You usually use the array_element function with the read_field_vals function. You use the array_element function to get a representation of an array element that is a structure or pointer to a structure. You then pass this representation to the read_field_vals function to get the values of fields inside the structure. For an example of how this is done, see Example 3–4 in Section 3.3.

The first argument of the array_element function is usually the result returned from the read_sym function.

Note

The read_sym, array_element, and read_field_vals functions are often used together to retrieve the values of an array of structures pointed to by a global pointer. (For more information about using these functions, see the description of the read_sym function in Section 3.2.27.)

For example:

```
if((ele = array_element(sz_softc, cntrl, &error)) == NULL){
  fprintf(stderr, "Couldn’t get %d’th element of sz_softc:\n, cntrl");
  fprintf(stderr, "%s\n", error);
}
```

3.2.4 Retrieving an Array Element Value

The array_element_val function returns the value of an array element. It returns the integer value if the data type of the array element is an integer data type. It returns the pointer value if the data type of the array element is a pointer data type.

This function returns TRUE if it is successful, FALSE otherwise. When the return value is FALSE, an error message is returned in an argument to the function.

This function has the following syntax:

Boolean array_element_val(DataStruct *sym* , int *i* , long * *ele_ret* , char ** *error*);

Argument	Input/Output	Description
sym	Input	Names the array
i	Input	Specifies the index of the element
ele_ret	Output	Returns the value of the pointer
error	Output	Returns a pointer to an error message if the return value is FALSE

You use the `array_element_val` function when the array element is of a basic C type. You also use this function if the array element is of a pointer type and the pointer value is what you actually want. This function returns a printable value. The first argument of the `array_element_val` function usually comes from the returned result of the `read_sym` function.

For example:

```
static char get_ele(array, i)
DataStruct array;
int i;
{
  char *error, ret;
  long val;

  if(!array_element_val(array, i, &val, &error)){
    fprintf(stderr, "Couldn't read array element:\n");
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  ret = val;
  return(ret);
}
```

3.2.5 Returning the Size of an Array

The `array_size` function returns the size of the specified array. This function has the following syntax:

unsigned int array_size(DataStruct *sym* , char ***error*);

Argument	Input/Output	Description
sym	Input	Names the array
error	Output	Returns a pointer to an error message if the return value is non-NULL

For example:

```
busses = read_sym("bus_list");

if((n = array_size(busses, &error)) == -1){
  fprintf(stderr, "Couldn't call array_size:\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.6 Casting a Pointer to a Data Structure

The `cast` function casts the pointer to a structure as a structure data type and returns the structure. This function has the following syntax:

Boolean cast(long *addr*, char * *type*, DataStruct * *ret_type*, char ** *error*);

Argument	Input/Output	Description
addr	Input	Specifies the address of the data structure you want returned
type	Input	Specifies the datatype of the data structure
ret_type	Output	Returns the name of the data structure
error	Output	Returns a pointer to an error message if the return value is FALSE

You usually use the `cast` function with the `read_field_vals` function. Given the address of a structure, you call the `cast` function to convert the pointer from the type `long` to the type `DataStruct`. Then, you pass the result to the `read_field_vals` function, as its first argument, to retrieve the values of data fields in the structure.

For example:

```
if(!cast(addr, "struct file", &fil, &error)){
  fprintf(stderr, "Couldn’t cast address to a file:\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.7 Checking Arguments Passed to an Extension

The `check_args` function checks the arguments passed to an extension or displays a help message. The function displays a help message when the user specifies the `–help` flag on the command line.

This function has the following syntax:

void check_args(int *argc*, char ** *argv*, char * *help_string*);

Argument	Input/Output	Description
argc	Input	Passes in the first argument to the command
argv	Input	Passes in the second argument to the command
help_string	Input	Specifies the help message to be displayed to the user

You should include the `check_args` function early in your extension to be sure that arguments are correct.

For example:

```
check_args(argc, argv, help_string);
if(!check_fields("struct sz_softc", fields, NUM_FIELDS, NULL)){
  field_errors(fields, NUM_FIELDS);
  quit(1);
}
```

3.2.8 Checking the Fields in a Structure

The `check_fields` function verifies that the specified function consists of the expected number of fields and that those fields have the correct data type. If the function is successful, TRUE is returned; otherwise, the error parts of the affected fields are filled in with errors, and FALSE is returned.

This function has the following syntax:

Boolean check_fields(char * *symbol*, FieldRec * *fields*, int *nfields*, char ** *hints*);

Argument	Input/Output	Description
symbol	Input	Names the structure to be checked
fields	Input	Describes the fields to be checked
nfields	Input	Specifies the size of the `fields` argument
hints	Input	Unused and should always be set to NULL

You should check the structure type using the `check_fields` function before using the `read_field_vals` function to read field values.

For example:

```
FieldRec fields[] = {
  {  ".sc_sysid", NUMBER, NULL, NULL },
  {  ".sc_aipfts", NUMBER, NULL, NULL },
  {  ".sc_lostarb", NUMBER, NULL, NULL },
  {  ".sc_lastid", NUMBER, NULL, NULL },
  {  ".sc_active", NUMBER, NULL, NULL }
};

check_args(argc, argv, help_string);
if(!check_fields("struct sz_softc", fields, NUM_FIELDS, NULL)){
  field_errors(fields, NUM_FIELDS);
  quit(1);
}
```

3.2.9 Setting the kdbx Context

The `context` function sets the context to `user` context or `proc` context. If the context is set to the `user` context, aliases defined in the extension affect user aliases.

This function has the following syntax:

void context(Boolean *user*);

Argument	Input/Output	Description
user	Input	Sets the context to user if TRUE or proc if FALSE

For example:

```
if(head) print(head);
context(True);
for(i=0;i<len;i++){
 :
```

3.2.10 Passing Commands to the dbx Debugger

The dbx function passes a command to the dbx debugger. The function has an argument, *expect_output*, that controls when it returns. If you set the expect_output argument to TRUE, the function returns after the command is sent, and expects the extension to read the output from dbx. If you set the *expect_output* argument to FALSE, the function waits for the command to complete execution, reads the acknowledgement from kdbx, and then returns.

void dbx(char * *command*, Boolean *expect_output*);

Argument	Input/Output	Description
command	Input	Specifies the command to be passed to dbx
expect_output	Input	Indicates whether the extension expects output and determines when the function returns

For example:

```
dbx(out, True);
if((buf = read_response(&status)) == NULL){
  print_status("main", &status);
  quit(1);
}
else {
  process_buf(buf);
  quit(0);
}
```

3.2.11 Dereferencing a Pointer

The deref_pointer function returns a representation of the object pointed to by a pointer. The function displays an error message if the *data* argument passed is not a valid address.

This function has the following syntax:

DataStruct deref_pointer(DataStruct *data*);

Argument	Input/Output	Description
data	Input	Names the data structure that is being dereferenced

For example:

```
structure = deref_pointer(struct_pointer);
```

3.2.12 Displaying the Error Messages Stored in Fields

The field_errors function displays the error messages stored in fields by the check_fields function. This function has the following syntax:

void field_errors(FieldRec * *fields*, int *nfields*);

Argument	Input/Output	Description
fields	Input	Names the fields that contain the error messages
nfields	Input	Specifies the size of the fields argument

For example:

```
if(!read_field_vals(proc, fields, NUM_FIELDS)){
  field_errors(fields, NUM_FIELDS);
  return(False);
}
```

3.2.13 Converting a Long Address to a String Address

The `format_addr` function converts a 64-bit address of type `long` into a 32-bit address of type `char`. This function has the following syntax:

extern char *format_addr(long *addr*, char * *buffer*);

Argument	Input/Output	Description
addr	Input	Specifies the address to be converted
buffer	Output	Returns the converted address and must be at least 12 characters long

Use this function to save space on the output line. For example, the 64-bit address `0xffffffff12345678` is converted into `v0x12345678`.

For example:

```
static Boolean prfile(DataStruct ele, long vn_addr, long socket_addr)
{
  char *error, op_buf[12], *ops, buf[256], address[12], cred[12], data[12];
  if(!read_field_vals(ele, fields, NUM_FIELDS)){
    field_errors(fields, NUM_FIELDS);
    return(False);
  }
  if((long) fields[1].data == 0) return(True);
  if((long) (fields[5].data) == 0) ops = "  *Null*  ";
  else if((long) (fields[5].data) == vn_addr) ops = "   vnops   ";
  else if((long) (fields[5].data) == socket_addr) ops = " socketops ";
  else format_addr((long) fields[5].data, op_buf);
  format_addr((long) struct_addr(ele), address);
  format_addr((long) fields[2].data, cred);
  format_addr((long) fields[3].data, data);
  sprintf(buf, "%s %s %4d %4d %s %s %s %6d   %s%s%s%s%s%s%s%s%s",
          address, get_type((int) fields[0].data), fields[1].data,
          fields[2].data, ops, cred, data, fields[6].data,
          ((long) fields[7].data) & FREAD ? " read" : ,
          ((long) fields[7].data) & FWRITE ? " write" : ,
          ((long) fields[7].data) & FAPPEND ? " append" : ,
          ((long) fields[7].data) & FNDELAY ? " ndelay" : ,
          ((long) fields[7].data) & FMARK ? " mark" : ,
          ((long) fields[7].data) & FDEFER ? " defer" : ,
          ((long) fields[7].data) & FASYNC ? " async" : ,
          ((long) fields[7].data) & FSHLOCK ? " shlck" : ,
          ((long) fields[7].data) & FEXLOCK ? " exlck" : );
  print(buf);
  return(True);
}
```

3.2.14 Freeing Memory

The `free_sym` function releases the memory held by a specified symbol. This function has the following syntax:

void free_sym(DataStruct *sym*);

Argument	Input/Output	Description
sym	Input	Names the symbol that is using memory that can be freed

For example:

```
free_sym(rec->data);
```

3.2.15 Passing Commands to the kdbx Debugger

The `krash` function passes a command to `kdbx` for execution. You specify the command you want passed to `kdbx` as the first argument to the `krash` function. The second argument allows you to pass quotation marks (""), apostrophes ('), and backslash characters (\) to `kdbx`. The function has an argument, *expect_output*, which controls when it returns. If you set the `expect_output` argument to TRUE, the function returns after the command is sent, and expects the extension to read the output from `dbx`. If you set the *expect_output* argument to FALSE, the function waits for the command to complete execution, reads the acknowledgement from `kdbx`, and then returns.

This function has the following syntax:

void krash(char * *command*, Boolean *quote*, Boolean *expect_output*);

Argument	Input/Output	Description
command	Input	Names the command to be executed
quote	Input	If set to TRUE causes the quote character, apostrophe, and backslash to be appropriately quoted so that they are treated normally, instead of as special characters
expect_output	Input	Indicates whether the extension expects output and determines when the function returns

For example:

```
do  {
     :
  if(doit){
    format(command, buf, type, addr, last, i, next);
    context(True);

    krash(buf, False, True);
    while((line = read_line(&status)) != NULL){
      print(line);
      free(line);
  }
  :
addr = next;
i++;
```

Suppose the preceding example is used to list the addresses of each node in the system mount table, which is a linked list. The following list describes the arguments to the `format` function in this case:

- The `command` argument contains the `dbx` command to be executed, such as `p` for `print`.
- The `buf` argument contains the full `dbx` command line; for example, `buf` might contain:

  ```
  p ((struct mount *) 0xffffffff8196db30).m_next
  ```

- The `type` argument contains the data type of each node in the list, as in `struct mount *`.

- The addr argument contains the address of the current node in the list; for example, the current node might be at address 0xffffffff8196db30.
- The last argument contains the address of the previous node in the list. In this case, last contains zero (0).
- The i argument is the current node's index. In this case, i contains 1.
- The next argument is the address of the next node in the list; for example, the next node might be at address 0xffffffff8196d050.

3.2.16 Getting the Address of an Item in a Linked List

The list_nth_cell function returns the address of one of the items in a linked list. This function has the following format:

Boolean list_nth_cell(long *addr*, char * *type*, int *n*,char * *next_field*, Boolean *do_check*, long * *val_ret*, **char **** *error*);

Argument	Input/Output	Description
addr	Input	Specifies the starting address of the linked list
type	Input	Specifies the data type of the item for which you are requesting an address
n	Input	Supplies a number indicating which list item's address is being requested
next_field	Input	Gives the name of the field that points to the next item in the linked list
do_check	Input	Determines whether kdbx checks the arguments to ensure that correct information is being sent (TRUE setting)
val_ret	Output	Returns the address of the requested list item
error	Output	Returns a pointer to an error message if the return value is FALSE

For example:

```
long root_addr, addr;
if (!read_sym_val("rootfs", NUMBER, &root_addr, &error)){
  :
}
```

```
if(!list_nth_cell(root_addr, "struct mount", i, "m_next", True, &addr,
                  &error)){
  fprintf(stderr, "Couldn’t get %d’th element of mount table\n", i);
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.17 Passing an Extension to kdbx

The `new_proc` function directs `kdbx` to execute a `proc` command with arguments specified in *args*. The *args* arguments can name a Digital-supplied extension or an extension that you create.

This function has the following syntax:

void new_proc(char * *args*, char ** *output_ret*);

Argument	Input/Output	Description
args	Input	Names the extensions to be passed to `kdbx`
output_ret	Output	Returns the output from the extension, if it is non-NULL

For example:

```
static void prmap(long addr)
{
  char cast_addr[36], buf[256], *resp;

  sprintf(cast_addr, "((struct\ vm_map_t\ *)\ 0x%p)", addr);
  sprintf(buf, "printf
          cast_addr);
  new_proc(buf, &resp);
  print(resp);
  free(resp);
}
```

3.2.18 Getting the Next Token as an Integer

The `next_number` function converts the next token in a buffer to an integer. The function returns TRUE if successful, or FALSE if there was an error.

This function has the following syntax:

Boolean next_number(char * *buf*, char ** *next*, long * *ret*);

Argument	Input/Output	Description
buf	Input	Names the buffer containing the value to be converted
next	Output	Returns a pointer to the next value in the buffer, if that value is non-NULL
ret	Output	Returns the integer value

For example:

```
resp = read_response_status();
next_number(resp, NULL, &size);
ret->size = size;
```

3.2.19 Getting the Next Token as a String

The `next_token` function returns a pointer to the next token in the specified pointer to a string. A token is a sequence of nonspace characters. This function has the following syntax:

char *next_token(char * *ptr*, int * *len_ret*, char ** *next_ret*);

Argument	Input/Output	Description
ptr	Input	Specifies the name of the pointer
len_ret	Output	Returns the length of the next token, if non-NULL
next_ret	Output	Returns a pointer to the first character after, but not included in the current token, if non-NULL

You use this function to extract words or other tokens from a character string. A common use, as shown in the example that follows, is to extract tokens from a string of numbers. You can then cast the tokens to a numerical data type, such as the `long` data type, and use them as numbers.

For example:

```
static long *parse_memory(char *buf, int offset, int size)
{
  long *buffer, *ret;
  int index, len;
  char *ptr, *token, *next;
  NEW_TYPE(buffer, offset + size, long, long *, "parse_memory");
  ret = buffer;
  index = offset;
  ptr = buf;
  while(index < offset + size){
    if((token = next_token(ptr, &len, &next)) == NULL){
      ret = NULL;
      break;
    }
    ptr = next;
    if(token[len - 1] == ':') continue;
    buffer[index] = strtoul(token, &ptr, 16);
    if(ptr != &token[len]){
      ret = NULL;
      break;
    }
    index++;
  }
  if(ret == NULL) free(buffer);
  return(ret);
}
```

3.2.20 Displaying a Message

The `print` function displays a message on the terminal screen. Because of the input and output redirection done by `kdbx`, all output to `stdout` from a `kdbx` extension goes to `dbx`. As a result, a `kdbx` extension cannot use normal C output functions such as `printf` and `fprintf(stdout,`...) to display information on the screen. Although the `fprintf(stderr,`...) function is still available, the recommended method is to first use the `sprintf` function to print the output into a character buffer and then use the `kdbx` library function `print` to display the contents of the buffer to the screen.

The `print` function automatically displays a newline character at the end of the output, it fails if it detects a newline character at the end of the buffer.

This function has the following format:

void print(char * *message*);

Argument	Input/Output	Description
message	Input	The message to be displayed

For example:

```
if(do_short){
  if(!check_fields("struct mount", short_mount_fields,
                   NUM_SHORT_MOUNT_FIELDS, NULL)){
    field_errors(short_mount_fields, NUM_SHORT_MOUNT_FIELDS);
    quit(1);
  }
  print("SLOT  MAJ  MIN TYPE                    DEVICE  MOUNT POINT");
}
```

3.2.21 Displaying Status Messages

The `print_status` function displays a status message that you supply and a status message supplied by the system. This function has the following format:

void print_status(char * *message*, Status * *status*);

Argument	Input/Output	Description
message	Input	Specifies the extension-defined status message
status	Input	Specifies the status returned from another library routine

For example:

```
if(status.type != OK){
  print_status("read_line failed", &status);
  quit(1);
}
```

3.2.22 Exiting from an Extension

The `quit` function sends a `quit` command to `kdbx`. This function has the following format:

void quit(int i);

Argument	Input/Output	Description
i	Input	The status at the time of the exit from the extension

For example:

```
if (!read_sym_val("vm_swap_head", NUMBER, &end, &error)) {
  fprintf(stderr, "Couldn't read vm_swap_head:\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.23 Reading the Values in Structure Fields

The `read_field_vals` function reads the value of fields in the specified structure. If this function is successful, then the data parts of the fields are filled in and TRUE is returned; otherwise, the error parts of the affected fields are filled in with errors and FALSE is returned.

This function has the following format:

Boolean read_field_vals(DataStruct *data*, FieldRec * *fields*, int *nfields*);

Argument	Input/Output	Description
data	Input	Names the structure that contains the field to be read
fields	Input	Describes the fields to be read
nfields	Input	Contains the size of the field array

For example:

```
if(!read_field_vals(pager, fields, nfields)){
  field_errors(fields, nfields);
  return(False);
}
```

3.2.24 Returning a Line of kdbx Output

The `read_line` function returns the next line of the output from the last `kdbx` command executed. If the end of the output is reached, this function returns NULL and a status of OK. If the status is something other than OK when the function returns NULL, an error occurred.

This function has the following format:

char *read_line(Status * *status*);

Argument	Input/Output	Description
status	Output	Contains the status of the request, which is OK for successful requests

For example:

```
while((line = read_line(&status)) != NULL){
  print(line);
  free(line);
}
```

3.2.25 Reading an Area of Memory

The `read_memory` function reads an area of memory starting at the address you specify and running for the number of bytes you specify. The `read_memory` function returns TRUE if successful and FALSE if there was an error.

This function has the following format:

Boolean read_memory(long *start_addr*, int *n*, char * *buf*, char ** *error*)

Argument	Input/Output	Description
start_addr	Input	Specifies the starting address for the read
n	Input	Specifies the number of bytes to read
buf	Output	Returns the memory contents
error	Output	Returns a pointer to an error message if the return value is FALSE

You can use this function to look up any type of value, however it is most useful for retrieving the value of pointers that point to other pointers.

For example:

```
start_addr = (long) ((long *)utask_fields[7].data + i-NOFILE_IN_U);
if(!read_memory(start_addr , sizeof(long *), (char *)&val1, &error) ||
  !read_memory((long)utask_fields[8].data , sizeof(long *), (char *)&val2,
     &error)){
  fprintf(stderr, "Couldn’t read_memory\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.26 Reading the Response to a kdbx Command

The `read_response` function reads the response to the last `kdbx` command entered. If any errors occurred, NULL is returned and the status argument is filled in.

This function has the following syntax:

char *read_response(Status * *status*);

Argument	Input/Output	Description
status	Output	Contains the status of the last `kdbx` command

For example:

```
if(!*argv) Usage();
command = argv;
if(size == 0){
  sprintf(buf, "print sizeof(*((%s) 0))", type);
  dbx(buf, True);

  if((resp = read_response(&status)) == NULL){
    print_status("Couldn’t read sizeof", &status);
    quit(1);
  }
  size = strtoul(resp, &ptr, 0);
  if(ptr == resp){

    fprintf(stderr, "Couldn’t parse sizeof(%s):\n", type);
    quit(1);
  }
```

```
  free(resp);
}
```

3.2.27 Reading Symbol Representations

The `read_sym` function returns a representation of the named symbol. This function has the following format:

DataStruct read_sym(char * *name*);

Argument	Input/Output	Description
name	Input	Names the symbol, which is normally a pointer to a structure or an array of structures inside the kernel

Often you use the result returned by the `read_sym` function as the input argument of the `array_element`, `array_element_val`, or `read_field_vals` function.

For example:

```
busses = read_sym("bus_list");
```

3.2.28 Reading a Symbol's Address

The `read_sym_addr` function reads the address of the specified symbol. This function has the following format:

Boolean read_sym_addr(char * *name*, long * *ret_val*, char ** *error*);

Argument	Input/Output	Description
name	Input	Names the symbol for which an address is required
ret_val	Output	Returns the address of the symbol
error	Output	Returns a pointer to an error message when the return status is FALSE

For example:

```
if(argc == 0) fil = read_sym("file");
if(!read_sym_val("nfile", NUMBER, &nfile, &error) ||
   !read_sym_addr("vnops", &vn_addr, &error) ||
   !read_sym_addr("socketops", &socket_addr, &error)){
  fprintf(stderr, "Couldn't read nfile:\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.29 Reading the Value of a Symbol

The `read_sym_val` function returns the value of the specified symbol. This function has the following format:

Boolean read_sym_val(char * *name*, int *type*, long * *ret_val*, char ** *error*);

Argument	Input/Output	Description
name	Input	Names the symbol for which a value is needed
type	Input	Specifies the data type of the symbol
ret_val	Output	Returns the value of the symbol
error	Output	Returns a pointer to an error message when the status is FALSE

You use the `read_sym_val` function to retrieve the value of a global variable. The value returned by the `read_sym_val` function has the type `long`, unlike the value returned by the `read_sym` function which has the type `DataStruct`.

For example:

```
if(argc == 0) fil = read_sym("file");
if(!read_sym_val("nfile", NUMBER, &nfile, &error) ||
   !read_sym_addr("vnops", &vn_addr, &error) ||
   !read_sym_addr("socketops", &socket_addr, &error)){
  fprintf(stderr, "Couldn't read nfile:\n");
  fprintf(stderr, "%s\n", error);
  quit(1);
}
```

3.2.30 Getting the Address of a Data Representation

The `struct_addr` function returns the address of a data representation. This function has the following format:

char *struct_addr(DataStruct *data*);

Argument	Input/Output	Description
data	Input	Specifies the structure for which an address is needed

For example:

```
if(bus_fields[1].data != 0){
  sprintf(buf, "Bus #%d (0x%p): Name - \"%s\"\tConnected to - \"%s\,
          i, struct_addr(bus), bus_fields[1].data, bus_fields[2].data);
  print(buf);
  sprintf(buf, "\tConfig 1 - %s\tConfig 2 - %s",
          addr_to_proc((long) bus_fields[3].data),
          addr_to_proc((long) bus_fields[4].data));
  print(buf);
  if(!prctlr((long) bus_fields[0].data)) quit(1);
  print();
}
```

3.2.31 Converting a String to a Number

The `to_number` function converts a string to a number. The function returns TRUE if successful, or FALSE if conversion was not possible.

This function has the following format:

Boolean to_number(char * *str*, long * *val*);

Argument	Input/Output	Description
str	Input	Contains the string to be converted
val	Output	Contains the numerical equivalent of the string

This function returns TRUE if successful, FALSE if conversion was not possible.

For example:

```
check_args(argc, argv, help_string);
if(argc < 5) Usage();
size = 0;
type = argv[1];
if(!to_number(argv[2], &len)) Usage();
addr = strtoul(argv[3], &ptr, 16);
if(*ptr != '\0'){
  if(!read_sym_val(argv[3], NUMBER, &addr, &error)){
    fprintf(stderr, "Couldn't read %s:\n", argv[3]);
    fprintf(stderr, "%s\n", error);
    Usage();
  }
}
```

3.3 Examples of kdbx Extensions

This section contains examples of the three types of extensions provided by the kdbx debugger:

- Extensions that use lists. Example 3–1 provides a C language template and Example 3–2 is the source code for the /var/kdbx/callout extension, which shows how to use linked lists in developing an extension.
- Extensions that use arrays. Example 3–3 provides a C language template and Example 3–4 is the source code for the /var/kdbx/file extension, which shows how to develop an extension using arrays.
- Extensions that use global symbols. Example 3–5 is the source code for the /var/kdbx/sum extensions, which shows how to pull global symbols from the kernel. A template is not provided because the means for pulling global symbols from a kernel can vary greatly, depending upon the desired output.

Example 3–1: Template Extension Using Lists

```
#include <stdio.h>
#include <kdbx.h>
static char *help_string =
```

Example 3–1: Template Extension Using Lists (cont.)

```
"<Usage info goes here>                                \\\n\ 1
";
FieldRec fields[] = {

  { ".<name of next field>", NUMBER, NULL, NULL }, 2

  <data fields>
};

#define NUM_FIELDS (sizeof(fields)/sizeof(fields[0]))

main(argc, argv)
int argc;
char **argv;
{
  DataStruct head;
  unsigned int next;
  char buf[256], *func, *error;

  check_args(argc, argv, help_string);
  if(!check_fields("<name of list structure>", fields, NUM_FIELDS, NULL)){ 3
    field_errors(fields, NUM_FIELDS);
    quit(1);
  }

  if(!read_sym_val("<name of list head>", NUMBER, (caddr_t *) &next, &error)){ 4
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  sprintf(buf, "<table header>"); 5
  print(buf);
  do {
    if(!cast(next, "<name of list structure>", &head, &error)){ 6
      fprintf(stderr, "Couldn't cast to a <struct>:\n"); 7
      fprintf(stderr, "%s:\n", error);
    }
    if(!read_field_vals(head, fields, NUM_FIELDS)){
      field_errors(fields, NUM_FIELDS);
      break;
    }
    <print data in this list cell> 8
    next = (int) fields[0].data;
  } while(next != 0);
  quit(0);
}
```

1. The help string is output by the `check_args` function if the user enters the `help extension_name` command at the `kdbx` prompt.The first line of the help string should be a one-line description of the extension. The rest should be a complete description of the arguments. Also, each line should end with the string `\\\n\`.

2. Every structure field to be extracted needs an entry. The first field is the name of the next extracted field; the second field is the type. The last two fields are for output and initialize to NULL.
3. Specifies the type of the list that is being traversed.
4. Specifies the variable that holds the head of the list.
5. Specifies the table header string.
6. Specifies the type of the list that is being traversed.
7. Specifies the structure type.
8. Extracts, formats, and prints the field information.

Example 3–2: Extension That Uses Linked Lists: callout.c

```
#include <stdio.h>
#include <errno.h>
#include <kdbx.h>

#define KERNEL
#include <sys/callout.h>

static char *help_string =
"callout - print the callout table                              \\\n\
    Usage : callout [cpu]                                       \\\n\
";

FieldRec processor_fields[] = {
  { ".calltodo.c_u.c_ticks", NUMBER, NULL, NULL },
  { ".calltodo.c_arg",  NUMBER, NULL, NULL },
  { ".calltodo.c_func", NUMBER, NULL, NULL },
  { ".calltodo.c_next", NUMBER, NULL, NULL },
  { ".lbolt",           NUMBER,  NULL, NULL },
  { ".state",           NUMBER,  NULL, NULL },
};

FieldRec callout_fields[] = {
  { ".c_u.c_ticks", NUMBER, NULL, NULL },
  { ".c_arg",  NUMBER, NULL, NULL },
  { ".c_func", NUMBER, NULL, NULL },
  { ".c_next", NUMBER, NULL, NULL },
};

#define NUM_PROCESSOR_FIELDS
(sizeof(processor_fields)/sizeof(processor_fields[0]))
#define NUM_CALLOUT_FIELDS (sizeof(callout_fields)/sizeof(callout_fields[0]))

main(int argc, char **argv)
{
  DataStruct processor_ptr, processor, callout;
```

Example 3–2: Extension That Uses Linked Lists: callout.c (cont.)

```
  long next, ncpus, ptr_val, i;
  char buf[256], *func, *error, arg[13];
  int cpuflag = 0, cpuarg = 0;

  long headptr;
  Status status;
  char *resp;

  if ( !(argc == 1 || argc == 2) ) {
    fprintf(stderr, "Usage: callout [cpu]\n");
    quit(1);
  }

  check_args(argc, argv, help_string);

  if (argc == 2)  {
    cpuflag = 1;
    errno = 0;
    cpuarg = atoi(argv[1]);
    if (errno != 0)
      fprintf(stderr, "Invalid argument value for the cpu number.\n");
  }

  if(!check_fields("struct processor", processor_fields, NUM_PROCESSOR_FIELDS,
NULL)){
    field_errors(processor_fields, NUM_PROCESSOR_FIELDS);
    quit(1);
  }

  if(!check_fields("struct callout", callout_fields, NUM_CALLOUT_FIELDS, NULL)){
    field_errors(callout_fields, NUM_CALLOUT_FIELDS);
    quit(1);
  }

  /* This gives the same result as "(kdbx) p processor_ptr" */
  if(!read_sym_addr("processor_ptr", &headptr, &error)){
    fprintf(stderr, "%s\n", error);
    quit(1);
  }

  /* get ncpus */
  if(!read_sym_val("ncpus", NUMBER, &ncpus, &error)){
    fprintf(stderr, "Couldn't read ncpus:\n");
    fprintf(stderr, "%s\n", error);
    quit(1);
  }

  for (i=0; i < ncpus; i++) {

    /* if user wants only one cpu and this is not the one, skip */
    if (cpuflag)
      if (cpuarg != i) continue;

    /* get the ith pointer (values) in the array */

    sprintf(buf, "set $hexints=0");
```

Example 3–2: Extension That Uses Linked Lists: callout.c (cont.)

```
    dbx(buf, False);
    sprintf(buf, "p \*(long \*)0x%lx", headptr+8*i);
    dbx(buf, True);
    if((resp = read_response(&status)) == NULL){
      print_status("Couldn't read value of processor_ptr[i]:", &status);
      quit(1);
    }
    ptr_val = strtoul(resp, (char**)NULL, 10);
    free(resp);

    if (! ptr_val)  continue;  /* continue if this slot is disabled */

    if(!cast(ptr_val, "struct processor", &processor, &error)){
      fprintf(stderr, "Couldn't cast to a processor:\n");
      fprintf(stderr, "%s:\n", error);
      quit(1);
    }

    if(!read_field_vals(processor, processor_fields, NUM_PROCESSOR_FIELDS)){
      field_errors(processor_fields, NUM_PROCESSOR_FIELDS);
      quit(1);
    }

    if (processor_fields[5].data == 0) continue;

    print("");
    sprintf(buf, "Processor:                        %10u", i);
    print(buf);
    sprintf(buf, "Current time (in ticks):          %10u",
processor_fields[4].data ); /*lbolt*/
    print(buf);


    /* for first element, we are interested in time only */

    print("");

    sprintf(buf, "        FUNCTION                   ARGUMENT    TICKS(delta)");
    print(buf);
    print(       "=============================    ============  ============");

    /* walk through the rest of the list */
    next = (long) processor_fields[3].data;
    while(next != 0) {
      if(!cast(next, "struct callout", &callout, &error)){
        fprintf(stderr, "Couldn't cast to a callout:\n");
        fprintf(stderr, "%s:\n", error);
      }
      if(!read_field_vals(callout, callout_fields, NUM_CALLOUT_FIELDS)){
        field_errors(callout_fields, NUM_CALLOUT_FIELDS);
        break;
      }
      func = addr_to_proc((long) callout_fields[2].data);
      format_addr((long) callout_fields[1].data, arg);
      sprintf(buf, "%-32.32s %12s %12d", func, arg,
     ((long)callout_fields[0].data & CALLTODO_TIME) -
```

Example 3–2: Extension That Uses Linked Lists: callout.c (cont.)

```
(long)processor_fields[4].data);
      print(buf);
      next = (long) callout_fields[3].data;
    }


  }  /* end of for */

  quit(0);

}  /* end of main() */
```

Example 3–3: Template Extensions Using Arrays

```
#include <stdio.h>
#include <kdbx.h>

static char *help_string =
"<Usage info>                                          \\\n\ 1
";

FieldRec fields[] = {
  <data fields> 2
};
#define NUM_FIELDS (sizeof(fields)/sizeof(fields[0]))
main(argc, argv)
int argc;
char **argv;
{
  int i, size;
  char *error, *ptr;
  DataStruct head, ele;
  check_args(argc, argv, help_string);

  if(!check_fields("<array element type>", fields, NUM_FIELDS, NULL)){ 3
    field_errors(fields, NUM_FIELDS);
    quit(1);
  }

  if(argc == 0) head = read_sym("<file>");  4

  if(!read_sym_val("<symbol containing size of array>", NUMBER, 5
     (caddr_t *) &size, &error) ||
    fprintf(stderr, "Couldn't read size:\n");
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
```

Example 3–3: Template Extensions Using Arrays (cont.)

```
  <print header> 6
  if(argc == 0){
    for(i=0;i<size;i++){
      if((ele = array_element(head, i, &error)) == NULL){
 fprintf(stderr, "Couldn't get array element\n");
 fprintf(stderr, "%s\n", error);
 return(False);
      }
      <print fields in this element> 7
    }
  }
}
```

1. The help string is output by the `check_args` function if the user enters the `help extension_name` command at the `kdbx` prompt. The first line of the help string should be a one-line description of the extension. The rest should be a complete description of the arguments. Also, each line should end with the string `\\\n\`.
2. Every structure field to be extracted needs an entry. The first field is the name of the next extracted field; the second field is the type. The last two fields are for output and initialize to NULL.
3. Specifies the type of the element in the array.
4. Specifies the variable containing the beginning address of the array.
5. Specifies the variable containing the size of the array. Note that reading variables is only one way to access this information. Other methods include the following:
   - Defining the array size with a `#define` macro call. If you use this method, you need to include the appropriate header file and use the macro in the extension.
   - Querying `dbx` for the array size as follows:

     ```
     dbx("print sizeof(array//sizeof(array[0]")
     ```

   - Hard coding the array size.
6. Specifies the string to be displayed as the table header.
7. Extracts, formats, and prints the field information.

Example 3–4: Extension That Uses Arrays: file.c

```
 #include <stdio.h>
#include <sys/fcntl.h>
#include <kdbx.h>
#include <nlist.h>
#define SHOW_UTT
#include <sys/user.h>
#define KERNEL_FILE
#include <sys/file.h>
#include <sys/proc.h>

static char *help_string =
"file - print out the file table                                          \\\n\
    Usage : file [addresses...]                                           \\\n\
    If no arguments are present, all file entries with non-zero reference \\\n\
    counts are printed. Otherwise, the file entries named by the addresses\\\n\
    are printed.                                                          \\\n\
";

char  buffer[256];

/* *** Implement addresses *** */

FieldRec fields[] = {
  { ".f_type", NUMBER, NULL, NULL },
  { ".f_count", NUMBER, NULL, NULL },
  { ".f_msgcount", NUMBER, NULL, NULL },
  { ".f_cred", NUMBER, NULL, NULL },
  { ".f_data", NUMBER, NULL, NULL },
  { ".f_ops", NUMBER, NULL, NULL },
  { ".f_u.fu_offset", NUMBER, NULL, NULL },
  { ".f_flag", NUMBER, NULL, NULL }
};

FieldRec fields_pid[] = {
  { ".pe_pid", NUMBER, NULL, NULL },
  { ".pe_proc", NUMBER, NULL, NULL },
};

FieldRec utask_fields[] = {
  { ".uu_file_state.uf_lastfile", NUMBER, NULL, NULL }, /* 0 */
  { ".uu_file_state.uf_ofile", ARRAY, NULL, NULL },     /* 1 */
  { ".uu_file_state.uf_pofile", ARRAY, NULL, NULL },    /* 2 */
  { ".uu_file_state.uf_ofile_of", NUMBER, NULL, NULL }, /* 3 */
  { ".uu_file_state.uf_pofile_of", NUMBER, NULL, NULL },/* 4 */
  { ".uu_file_state.uf_of_count", NUMBER, NULL, NULL }, /* 5 */
};

#define NUM_FIELDS (sizeof(fields)/sizeof(fields[0]))
#define NUM_UTASK_FIELDS (sizeof(utask_fields)/sizeof(utask_fields[0]))

static char *get_type(int type)
{
  static char buf[5];

  switch(type){
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
  case 1: return("file");
  case 2: return("sock");
  case 3: return("npip");
  case 4: return("pipe");
  default:
    sprintf(buf, "*%3d", type);
    return(buf);
  }
}

long vn_addr, socket_addr;
int proc_size; /* will be obtained from dbx */

static Boolean prfile(DataStruct ele)
{
  char *error, op_buf[12], *ops, buf[256], address[12], cred[12], data[12];

  if(!read_field_vals(ele, fields, NUM_FIELDS)){
    field_errors(fields, NUM_FIELDS);
    return(False);
  }
  if((long) fields[1].data == 0) return(True);
  if((long) (fields[5].data) == 0) ops = " *Null*";
  else if((long) (fields[5].data) == vn_addr) ops = "  vnops";
  else if((long) (fields[5].data) == socket_addr) ops = "sockops";
  else format_addr((long) fields[5].data, op_buf);
  format_addr((long) struct_addr(ele), address);
  format_addr((long) fields[3].data, cred);
  format_addr((long) fields[4].data, data);
  sprintf(buf, "%s %s %4d %4d %s %11s %11s %6d%s%s%s%s%s%s%s%s%s",
   address, get_type((int) fields[0].data), fields[1].data,
   fields[2].data, ops, data, cred, fields[6].data,
   ((long) fields[7].data) & FREAD ? " r" : "",
   ((long) fields[7].data) & FWRITE ? " w" : "",
   ((long) fields[7].data) & FAPPEND ? " a" : "",
   ((long) fields[7].data) & FNDELAY ? " nd" : "",
   ((long) fields[7].data) & FMARK ? " m" : "",
   ((long) fields[7].data) & FDEFER ? " d" : "",
   ((long) fields[7].data) & FASYNC ? " as" : "",
   ((long) fields[7].data) & FSHLOCK ? " sh" : "",
   ((long) fields[7].data) & FEXLOCK ? " ex" : "");
  print(buf);
  return(True);
}

static Boolean prfiles(DataStruct fil, int n)
{
  DataStruct ele;
  char *error;

  if((ele = array_element(fil, n, &error)) == NULL){
    fprintf(stderr, "Couldn't get array element\n");
    fprintf(stderr, "%s\n", error);
    return(False);
  }
  return(prfile(ele));
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
}

static void Usage(void){
  fprintf(stderr, "Usage : file [addresses...]\n");
  quit(1);
}

main(int argc, char **argv)
{
  int i;
  long nfile, addr;
  char *error, *ptr, *resp;
  DataStruct fil;
  Status status;

  check_args(argc, argv, help_string);
  argv++;
  argc--;

  if(!check_fields("struct file", fields, NUM_FIELDS, NULL)){
    field_errors(fields, NUM_FIELDS);
    quit(1);
  }
  if(!check_fields("struct pid_entry", fields_pid, 2, NULL)){
    field_errors(fields, 2);
    quit(1);
  }
  if(!check_fields("struct utask", utask_fields,  NUM_UTASK_FIELDS, NULL)){
    field_errors(fields, NUM_UTASK_FIELDS);
    quit(1);
  }

  if(!read_sym_addr("vnops", &vn_addr, &error) ||
     !read_sym_addr("socketops", &socket_addr, &error)){
    fprintf(stderr, "Couldn't read vnops or socketops:\n");
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  print("Addr        Type  Ref  Msg Fileops      F_data        Cred Offset
Flags");
  print("=========== ====  ===  === ======= =========== =========== ======
=====");
  if(argc == 0){
    /*
     * New code added to access open files in processes, in
     * the absence of static file table, file, nfile, etc..
     */

    /*
     * get the size of proc structure
     */
    sprintf(buffer, "set $hexints=0");
    dbx(buffer, False);
    sprintf(buffer, "print sizeof(struct proc)");
    dbx(buffer, True);
    if((resp = read_response(&status)) == NULL){
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
      print_status("Couldn’t read sizeof proc", &status);
      proc_size = sizeof(struct proc);
    }
    else
      proc_size = strtoul(resp, (char**)NULL, 10);
    free(resp);

    if ( get_all_open_files_from_active_processes() ) {
      fprintf(stderr, "Couldn’t get open files from processes:\n");
      quit(1);
    }
  }
  else {
    while(*argv){
      addr = strtoul(*argv, &ptr, 16);
      if(*ptr != ’\0’){
 fprintf(stderr, "Couldn’t parse %s to a number\n", *argv);
 quit(1);
      }
      if(!cast(addr, "struct file", &fil, &error)){
 fprintf(stderr, "Couldn’t cast address to a file:\n");
 fprintf(stderr, "%s\n", error);
 quit(1);
      }
      if(!prfile(fil))
 fprintf(stderr, "Continuing with next file address.\n");
      argv++;
    }
  }
  quit(0);
}

/*
 * Figure out the location of the utask structure in the supertask
 * #define proc_to_utask(p) (long)(p+sizeof(struct proc))
 */


/*
 * Figure out if this a system with the capability of
 * extending the number of open files per process above 64
 */
#ifdef NOFILE_IN_U
#  define OFILE_EXTEND
#else
#  define NOFILE_IN_U NOFILE
#endif

/*
 * Define a generic NULL pointer
 */
#define NIL_PTR(type) (type *) 0x0

get_all_open_files_from_active_processes()
{
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
 long pidtab_base;       /* Start address of the process table          */
 long npid;              /* Number of processes in the process table    */
 char *error;

 if (!read_sym_val("pidtab",  NUMBER, &pidtab_base, &error) ||
     !read_sym_val("npid", NUMBER, &npid, &error) ){
   fprintf(stderr, "Couldn't read pid or npid:\n");
   fprintf(stderr, "%s\n", error);
   quit(1);

 }

 if ( check_procs (pidtab_base, npid) )
   return(0);
 else
   return(1);
}


check_procs(pidtab_base, npid)
  long pidtab_base;
  long npid;
{
  int i, index, first_file;
  long addr;
  DataStruct pid_entry_struct, pid_entry_ele, utask_struct, fil;
  DataStruct ofile, pofile;
  char *error;
  long addr_of_proc, start_addr, val1, fp, last_fp;
  char  buf[256];


  /*
   * Walk the pid table
   */
  pid_entry_struct = read_sym("pidtab");

  for (index = 0; index < npid; index++)
  {
    if((pid_entry_ele = array_element(pid_entry_struct, index, &error))==NULL){
      fprintf(stderr, "Couldn't get pid array element %d\n", index);
      fprintf(stderr, "%s\n", error);
      continue;
    }
    if(!read_field_vals(pid_entry_ele, fields_pid, 2)) {
      fprintf(stderr, "Couldn't get values of pid array element %d\n", index);
      field_errors(fields_pid, 2);
      continue;
    }
    addr_of_proc = (long)fields_pid[1].data;
    if (addr_of_proc == 0)
      continue;
    first_file = True;
    addr = addr_of_proc + proc_size;

    if(!cast(addr, "struct utask", &utask_struct, &error)){
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
      fprintf(stderr, "Couldn’t cast address to a utask (bogus?):\n");
      fprintf(stderr, "%s\n", error);
      continue;
    }
    if(!read_field_vals(utask_struct, utask_fields, 3)) {
      fprintf(stderr, "Couldn’t read values of utask:\n");
      field_errors(fields_pid, 3);
      continue;
    }
    addr = (long) utask_fields[1].data;
    if (addr == NULL)
      continue;

    for(i=0;i<=(int)utask_fields[0].data;i++){
      if(i>=NOFILE_IN_U){
 if (utask_fields[3].data == NULL)
   continue;
 start_addr = (long)((long *)utask_fields[3].data + i-NOFILE_IN_U) ;
 if(!read_memory(start_addr , sizeof(struct file *), (char *)&val1,
&error)) {
   fprintf(stderr,"Start addr:0x%lx bytes:%d\n", start_addr, sizeof(long
*));
   fprintf(stderr, "Couldn’t read memory for extn files: %s\n", error);
   continue;
 }
      }
      else {
 ofile = (DataStruct) utask_fields[1].data;
 pofile = (DataStruct) utask_fields[2].data;
      }
      if (i < NOFILE_IN_U)
 if(!array_element_val(ofile, i, &val1, &error)){
   fprintf(stderr,"Couldn’t read %d’th element of ofile|pofile:\n", i);
   fprintf(stderr, "%s\n", error);
   continue;
 }
      fp = val1;
      if(fp == 0) continue;
      if(fp == last_fp) continue; /* eliminate duplicates */
      last_fp = fp;
      if(!cast(fp, "struct file", &fil, &error)){
 fprintf(stderr, "Couldn’t cast address to a file:\n");
 fprintf(stderr, "%s\n", error);
 quit(1);
      }
      if (first_file) {
 sprintf(buf, "[Process ID: %d]",  fields_pid[0].data);
 print(buf);
 first_file = False;
      }
      if(!prfile(fil))
 fprintf(stderr, "Continuing with next file address.\n");
    }
  } /* for loop */

  return(True);
```

Example 3–4: Extension That Uses Arrays: file.c (cont.)

```
} /* end */
```

Example 3–5: Extension That Uses Global Symbols: sum.c

```
#include <stdio.h>
#include <kdbx.h>

static char *help_string =
"sum - print a summary of the system                                      \\\n\
    Usage : sum                                                           \\\n\
";

static void read_var(name, type, val)
char *name;
int type;
long *val;
{
  char *error;
  long n;

  if(!read_sym_val(name, type, &n, &error)){
    fprintf(stderr, "Reading %s:\n", name);
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  *val = n;
}

main(argc, argv)
int argc;
char **argv;
{
  DataStruct utsname, cpup, time;
  char buf[256], *error, *resp, *sysname, *release, *version, *machine;
  long avail, secs, tmp;

  check_args(argc, argv, help_string);
  read_var("utsname.nodename", STRING, &resp);
  sprintf(buf, "Hostname : %s", resp);
  print(buf);
  free(resp);
  read_var("ncpus", NUMBER, &avail);
/*
 * cpup no longer exists, emmulate platform_string(),
 * a.k.a. get_system_type_string().
  read_var("cpup.system_string", STRING, &resp);
 */
  read_var("rpb->rpb_vers", NUMBER, &tmp);
  if (tmp < 5)
```

Example 3–5: Extension That Uses Global Symbols: sum.c (cont.)

```
      resp = "Unknown System Type";
  else
      read_var(
"(char *)rpb + rpb->rpb_dsr_off + "
"((struct rpb_dsr *)"
 " ((char *)rpb + rpb->rpb_dsr_off))->rpb_sysname_off + sizeof(long)",
        STRING, &resp);
  sprintf(buf, "cpu: %s\tavail: %d", resp, avail);
  print(buf);
  free(resp);
  read_var("boottime.tv_sec", NUMBER, &secs);
  sprintf(buf, "Boot-time:\t%s", ctime(&secs));
  buf[strlen(buf) - 1] = '\0';
  print(buf);
  read_var("time.tv_sec", NUMBER, &secs);
  sprintf(buf, "Time:\t%s", ctime(&secs));
  buf[strlen(buf) - 1] = '\0';
  print(buf);
  read_var("utsname.sysname", STRING, &sysname);
  read_var("utsname.release", STRING, &release);
  read_var("utsname.version", STRING, &version);
  read_var("utsname.machine", STRING, &machine);
  sprintf(buf, "Kernel : %s release %s version %s (%s)", sysname, release,
   version, machine);
  print(buf);
  quit(0);
}
```

3.4 Compiling Custom Extensions

After you have written the extension, you need to compile it. To compile the extension, enter the following command:

% **cc -o test test.c -lkdbx**

This `cc` command compiles an extension named `test.c`. The `kdbx.a` library is linked with the extensions, as specified by the `–l` flag. The output from this command is named `test`, as specified by the `–o` flag.

Once the extension compiles successfully, you should test it and, if necessary, debug it as described in Section 3.5.

When the extension is ready for use, place it in a directory that is accessible to other users. Digital UNIX extensions are located in the `/var/kdbx` directory.

The following example shows how to invoke the test extension from within the kdbx debugger:

```
# kdbx -k /vmunix
dbx version 3.12.1
 Type 'help' for help.

(kdbx) test
Hostname : system.dec.com
cpu: DEC3000 - M500     avail: 1
Boot-time:      Fri Nov  6 16:09:10 1992
Time:   Mon Nov  9 10:51:48 1992
Kernel : OSF1 release 1.2 version 1.2 (alpha)
(kdbx)
```

3.5 Debugging Custom Extensions

The kdbx debugger and the dbx debugger include the capability to communicate with each other using two named pipes. The task of debugging an extension is easier if you use a workstation with two windows or two terminals. In this way, you can dedicate one window or terminal to the kdbx debugger and one window or terminal to the dbx debugger. However, you can debug an extension from a single terminal. This section explains how to begin your kdbx and dbx sessions when you have two windows or terminals and when you have a single terminal. The examples illustrate debugging the test extension that was compiled in Section 3.4.

If you are using a workstation with two windows or have two terminals, perform the following steps to set up your kdbx and dbx debugging sessions:

1. Open two sessions: one running kdbx on the running kernel and the other running dbx on the source file for the custom extension test as follows:

   Begin the kdbx session:

   ```
   # kdbx -k /vmunix
   dbx version 3.12.1
   Type 'help' for help.

   stopped at [thread_block:1440 ,0xfffffc00002de5b0]   Source not available
   ```

   Begin the dbx session:

```
# dbx test
dbx version 3.12.1
Type 'help' for help.

(dbx)
```

2. Set up kdbx and dbx to communicate with each other. In the kdbx session, enter the procpd alias to create the files /tmp/pipein and /tmp/pipeout as follows:

   ```
   (kdbx) procpd
   ```

   The file pipein directs output from the dbx session to the kdbx session. The file pipeout directs output from the kdbx session to the dbx session.

3. In the dbx session, enter the run command to execute the test extension in the kdbx session, specifying the files /tmp/pipein and /tmp/pipeout on the command line as follows:

   ```
   (dbx) run < /tmp/pipeout > /tmp/pipein
   ```

4. As you step through the extension in the dbx session, you see the results of any action in the kdbx session. At this point, you can use the available dbx commands and options.

If you are using one terminal, perform the following steps to set up your kdbx and dbx sessions:

1. Issue the following command to invoke kdbx with the debugging environment:

   ```
   # echo 'procpd' | kdbx -k /vmunix &
   dbx version 3.12.1
   Type 'help' for help.

   stopped at  [thread_block:1403 ,0xfffffc000032d860]    Source not available

   #
   ```

2. Invoke the dbx debugger as follows:

   ```
   # dbx test
   dbx version 3.12.1
   Type 'help' for help.

   (dbx)
   ```

3. As you step through the extension in the dbx session, you see the results of any action in the kdbx session. At this point, you can use the available dbx commands and options.

4

Managing Crash Dumps

When a Digital UNIX system crashes, it writes all or part of physical memory to disk. This information is called a crash dump. During the reboot process, the system moves the crash dump into a file and copies the kernel executable image to another file. Together, these files are the crash dump files. You can use the information in the crash dump files to help you to determine the cause of the system crash.

To ensure that you can analyze crash dump files following a system crash, you must understand how crash dump files are created. You must reserve space on disks for the crash dump and crash dump files. The amount of space you reserve depends on your system configuration and the type of crash dump you want the system to perform.

This chapter gives the following information to help you manage crash dumps and crash dump files:

- How the system saves a crash dump to disk partitions at the time of the crash (Section 4.1)
- The types of crash dumps available and the procedures for choosing what type of crash dump is created (Section 4.2)
- Guidelines for deciding how much disk space to allow for crash dumps and procedures for controlling where crash dumps are written (Section 4.3)
- How the system moves the crash dump and the executable kernel image into crash dump files at system reboot time and a description of how crash logging is performed at system reboot time (Section 4.4)
- Guidelines for deciding how much disk space to allocate for crash dump files and the procedure for changing the location to which crash dump files are written (Section 4.5)
- Information about compressing and uncompressing crash dump files (Section 4.6)
- How to cause a hung system to crash so that a crash dump is created (Section 4.7)

For information about analyzing the contents of crash dump files, see Chapter 5.

4.1 Crash Dump Creation

When the system creates a crash dump, it writes the dump to the swap partitions. The system uses the swap partitions because the information stored in those partitions has meaning only for a running system. Once the system crashes, the information is useless and can be safely overwritten.

Before the system writes a crash dump, it determines how the dump fits into the swap partitions. The following list describes how the system determines where to write the crash dump:

1. If the crash dump fits in the primary swap partition, (`swap1` in the `/etc/fstab` file) the system writes the dump to the end of that partition. The system writes the dump as far toward the end of the partition as possible, leaving the beginning of the partition available for swapping done at system reboot time.
2. If the crash dump is too large for the primary swap partition, the system writes the crash dump to the secondary swap partitions (`swap2` in the `/etc/fstab` file.) You can have multiple secondary swap partitions on multiple devices.
3. If the crash dump is too large for the secondary swap partitions, the system writes the crash dump to the secondary swap partitions until those partitions are full. It then writes the remaining crash dump information to end of the primary swap partition, possibly filling that partition.

Note

If the aggregate size of all the swap partitions is too small to contain the crash dump, the system creates no crash dump.

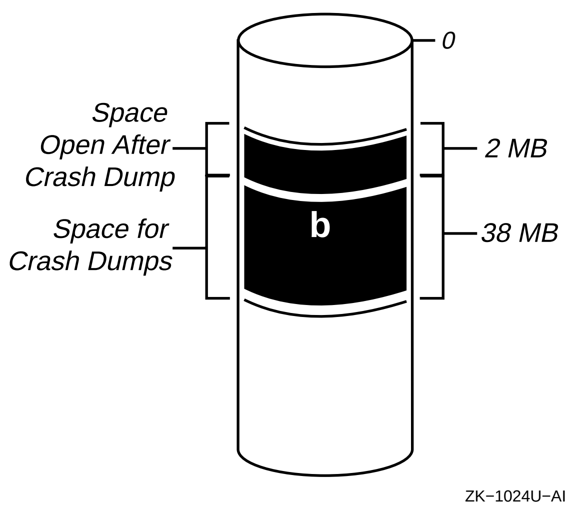
Each crash dump contains a header, which the system always writes to the end of the primary swap partition. The header contains information about the size of the dump and where the dump is stored. This information allows the system to find and save the dump at system reboot time.

You can configure the system so that it fills the secondary swap partitions with dump information before writing any information (except the dump header) to the primary swap partition. The attribute that you use to configure where crash dumps are written first is the dump_sp_threshold attribute.

The value in the dump_sp_threshold attribute indicates the amount of space you normally want available for swapping as the system reboots. By default, this attribute is set to 4096 blocks, meaning that the system attempts to leave 2 MB of disk space open in the primary swap partition after the dump is written.

Figure 4–1 shows the default setting of the dump_sp_threshold attribute for a 40 MB swap partition.

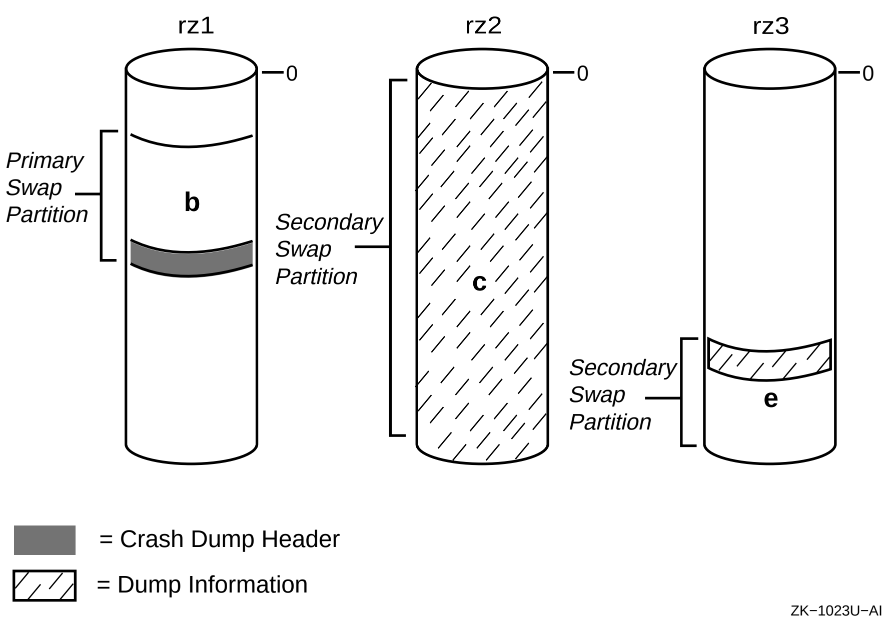
Figure 4–1: Default dump_sp_threshold Attribute Setting



The system can write 38 MB of dump information to the primary swap partion shown in Figure 4–1. Therefore, a 30 MB dump fits on the primary swap partition and is written to that partition. However, a 40 MB dump is too large; the system writes the crash dump header to the end of the primary swap partition and writes the rest of the crash dump to secondary swap partitions.

Setting the dump_sp_threshold attribute to a high value causes the system to fill the secondary swap partitions before it writes dump information to the primary swap partion. For example, if you set the dump_sp_threshold attribute to a value that is equal to the size of the primary swap partition, the system fills the secondary swap partitions first. (Setting the dump_sp_threshold attribute is described in Section 4.3.3.) Figure 4–2 illustrates how a crash dump is written to secondary swap partitions on multiple devices.

Figure 4–2: Crash Dump Written to Multiple Devices



If the crash dump fills partition e in Figure 4–2, the system writes the remaining crash dump information to the end of the primary swap partition. Note that the system fills as much of the primary swap partition as is necessary to store the entire dump. The dump is written to the end of the primary swap partition to attempt to protect it from system swapping. However, the dump can fill the entire primary swap partition and might be corrupted by swapping that occurs as the system reboots.

4.2 Choosing the Contents of Crash Dumps

Crash dumps are partial (the default) or full. Normally, partial crash dumps provide the information that you need to determine the cause of a crash. However, you might want the system to generate full crash dumps if you have a recurring crash problem and partial crash dumps have not been helpful in finding the cause of the crash.

A partial crash dump contains the following:

- The crash dump header
- A copy of part of physical memory

  The system writes the part of physical memory believed to contain significant information at the time of the system crash. By default, the system omits user page table entries.

A full crash dump contains the following:

- The crash dump header
- A copy of the entire contents of physical memory at the time of the crash

As explained in the sections that follow, you can control the contents of crash dumps in the following two ways:

- By overriding the default so that the system writes user page table entries to partial crash dumps
- By selecting partial or full crash dumps

4.2.1 Including User Page Tables in Partial Crash Dumps

By default, the system omits user page tables from partial crash dumps. These tables do not normally help you determine the cause of a crash and omitting them reduces the size of crash dumps and crash dump files.

If you want the system to include user page tables in partial crash dumps, set the value of the `dump-user-pte-pages` attribute to 1. The `dump-user-pte-pages` attribute is in the `vm` subsystem. The following example shows the command you issue to set this attribute:

```
# sysconfig -r vm dump-user-pte-pages = 1
```

The `sysconfig` command changes the value of system attributes for the currently running kernel. To store the new value of the `dump-user-pte-pages` attribute in the `sysconfigtab` database, modify that database using the `sysconfigdb` command. For information about the `sysconfigtab` database and the `sysconfigdb` command, see the *System Administration* manual and the `sysconfigdb`(8) reference page.

To return to the system default of not writing user page tables to partial crash dumps, set the value of the dump-user-pte-pages attribute to 0 (zero).

4.2.2 Selecting Partial or Full Crash Dumps

By default, the system generates partial crash dumps. If you want the system to generate full crash dumps, you can modify the default behavior in the following ways:

- Specify the d flag to the boot_osflags console environment variable.

  To set this console environment variable, shut down and halt your system. At the console prompt, enter the following command:

  ```
  >>> set boot_osflags d
  ```

  The boot_osflags variable controls other boot options, such as whether the system boots to single-user mode or multiuser mode; therefore, use care when setting this variable. For more information about boot_osflags, see the *System Administration* manual.

- Set the kernel's partial_dump variable to 0 (zero) using the dbx debugger as follows:

  ```
  (dbx) a partial_dump = 0
  ```

To return to partial crash dumps, remove the d flag from the boot_osflags environment variable or set the partial_dump variable to 1.

4.3 Planning Crash Dump Space

Because crash dumps are written to the swap partitions on your system, you allow space for crash dumps by adjusting the size of your swap partitions. For information about modifying the size of swap partitions, see the *System Administration* manual and the *Installation Guide*.

Note

Be sure to list all swap partitions in the /etc/fstab file. The savecore command, which copies the crash dump from swap partitions to a file, uses the information in the /etc/fstab file to find the swap partitions. If you omit a swap partition from /etc/fstab, the savecore command might be unable to find the omitted partition.

The sections that follow give guidelines for estimating the amount of space required for partial and full crash dumps. In addition, setting the dump_sp_threshold attribute is described.

4.3.1 Estimating the Size of Partial Crash Dumps

Normally, a partial crash dump contains only a part of physical memory, so you allocate less disk space to saving a partial crash dump than you allocate for a full crash dump. The amount of space required to save a partial crash dump varies, depending on the level of system activity. For example, suppose your system has 128 MB of memory, but your peak system activity level is low (never uses more than 60 MB of memory.) In this case, you might allow 70 MB of disk space for storing crash dumps.

If your swap partitions are too small to store a partial crash dump, the system creates no crash dump. Therefore, overestimate the amount of space you need and adjust the amount of space you allocate to saving crash dumps, if necessary, after your system creates a few crash dumps.

Because crash dumps are about the same size as crash dump files, you can determine how large a crash dump was by examining the size of the resulting crash dump file. For example, to determine how large the first crash dump file created by your system is, issue the following command:

```
# ls -s /var/adm/crash/vmcore.0
20480 vmcore.0
```

This command displays the number of 512-byte blocks occupied by the crash dump file. In this case, the file occupies 20,480 blocks, so you know that the crash dump written to the swap partitions also occupied about 20,480 blocks. Be sure to use the `ls -s` command to display the size of crash dump files. The size that the `ls -l` command displays is incorrect.

The `ls -l` command includes file “holes” in the size of the crash dump file. (See Section 4.6 for more information.)

In some cases, a system contains so much active memory that it cannot store a crash dump on a single disk. For example, suppose your system contains 2 GB of memory and system activity level is high (uses most of memory). Crash dumps for this system are too large to fit on a single device. To cause crash dumps to spread across multiple disks, set the `dump_sp_threshold` attribute to a high value, as described in Section 4.3.3, and create secondary swap partitions on several disks. The system automatically writes dumps that are too large to fit in the primary swap partition to secondary swap partitions. The *System Administration* manual describes configuring swap space.

4.3.2 Estimating the Size of Full Crash Dumps

Full crash dumps provide you the maximum information about the system at the time of the crash. However, this type of crash dump occupies a large amount of disk space. If you intend to save full crash dumps, you need to create swap partitions equal to the size of memory, plus 1 additional block for the crash dump header. For example, if your system has 128 MB of memory, your swap partitions must provide at least 129 MB of disk space, with at least 1 block of disk space in the primary swap partition to store the crash dump header.

If your system contains a large amount (2 GB, for example) of memory, it might need to spread crash dumps across multiple disks. To cause crash dumps to spread across multiple disks, set the `dump_sp_threshold` attribute to a high value, as described in Section 4.3.3, and create secondary swap partitions on several disks. The system automatically writes dumps that are too large to fit in the primary swap partition to secondary swap partitions. The *System Administration* manual describes configuring swap space.

If you chose to have the system perform a full dump when it crashes and your swap partitions are too small to store a full dump, the system performs a partial dump.

4.3.3 Adjusting the Primary Swap Partition’s Crash Dump Threshold

To configure your system so that it writes crash dumps to secondary swap partitions before the primary swap partition, use the `dump_sp_threshold` attribute. As described in Section 4.1, the value you assign to this attribute indicates the amount of space that you normally want available for system swapping after a system crash.

To adjust the dump_sp_threshold attribute, issue the sysconfig command. For example, suppose your primary swap partition is 40 MB. To raise the value so that the system writes crash dumps to secondary partitions, issue the following command:

```
# sysconfig -r generic dump_sp_threshold=20480
```

In the preceding example, the dump_sp_threshold attribute, which is in the generic subsystem, is set to 20,480 512-byte blocks (40 MB). In this example, the system attempts to leave the entire primary swap partition open for system swapping. The system automatically writes the crash dump to secondary swap partitions and the crash dump header to the end of the primary swap partition.

The sysconfig command changes the value of system attributes for the currently running kernel. To store the new value of the dump_sp_threshold attribute in the sysconfigtab database, modify that database using the sysconfigdb command. For information about the sysconfigtab database and the sysconfigdb command, see the *System Administration* manual and the sysconfigdb(8) reference page.

4.4 Crash Dump File Creation and Crash Dump Logging

After a system crash, you normally reboot your system by issuing the boot command at the console prompt. During a system reboot, the /sbin/init.d/savecore script invokes the savecore command. This command moves crash dump information from the swap partitions into a file and copies the kernel that was running at the time of the crash into another file. You can analyze these files to help you determine the cause of a crash. The savecore command also logs the crash in system log files.

You can invoke the savecore command from the command line. For information about the command syntax, see the savecore(8) reference page.

4.4.1 Crash Dump File Creation

When the savecore command begins running during the reboot process, it determines whether a crash dump occurred and whether the file system contains enough space to save it. (The system saves no crash dump if you shut it down and reboot it; that is, the system saves a crash dump only when it crashes.)

If a crash dump exists and the file system contains enough space to save the crash dump files, the savecore command moves the crash dump and a copy of the kernel into files in the default crash directory, /var/adm/crash. (You can modify the location of the crash directory, as described in Section 4.5.) The savecore command stores the kernel image in a file named vmunix.*n*, and it stores the contents of physical memory in a file named vmcore.*n*.

The *n* variable specifies the number of the crash. The number of the crash is recorded in the bounds file in the crash directory. After the first crash, the savecore command creates the bounds file and stores the number 1 in it. The command increments that value for each succeeding crash.

The savecore command runs early in the reboot process so that little or no system swapping occurs before the command runs. This practice helps ensure that crash dumps are not corrupted by swapping.

4.4.2 Crash Dump Logging

Once the savecore command writes the crash dump files, it performs the following steps to log the crash in system log files:

1. Writes a reboot message to the /var/adm/syslog/auth.log file. If the system crashed due to a panic condition, the panic string is included in the log entry.

   You can cause the savecore command to write the reboot message to another file by modifying the auth facility entry in the syslog.conf file. If you remove the auth entry from the syslog.conf file, the savecore command does not save the reboot message.

2. Attempts to save the kernel message buffer from the crash dump. The kernel message buffer contains messages created by the kernel that crashed. These messages might help you determine the cause of the crash.

   The savecore command saves the kernel message buffer in the /var/adm/crash/msgbuf.savecore file, by default. You can change the location to which savecore writes the kernel message buffer by modifying the msgbuf.err entry in the /etc/syslog.conf file. If you remove the msgbuf.err entry from the /etc/syslog.conf file, savecore does not save the kernel message buffer.

Later in the reboot process, the syslogd daemon starts up, reads the contents of the msgbuf.err file, and moves those contents into the /var/adm/syslog/kern.log file, as specified in the /etc/syslog.conf file. The syslogd daemon then deletes the msgbuf.err file. For more information about how system logging is performed, see the *System Administration* manual and the syslogd(8) reference page.

3. Attempts to save the binary event buffer from the crash dump. The binary event buffer contains messages that can help you identify the problem that caused the crash, particularly if the crash was due to a hardware error.

   The savecore command saves the binary event buffer in the /usr/adm/crash/binlogdumpfile file by default. You can change the location to which savecore writes the binary event buffer by modifying the dumpfile entry in the /etc/binlog.conf file. If you remove the dumpfile entry from the /etc/binlog.conf file, savecore does not save the binary event buffer.

   Later in the reboot process the binlogd daemon starts up, reads the contents of the /usr/adm/crash/binlogdumpfile file, and moves those contents into the /usr/adm/binary.errlog file, as specified in the /etc/binlog.conf file. The binlogd daemon then deletes the binlogdumpfile file. For more information about how binary error logging is performed, see the *System Administration* manual and the binlogd(8) reference page.

4.5 Planning and Allocating File System Space for Crash Dump Files

The size of crash dump files varies, depending on whether you use partial crash dumps or full crash dumps. In the case of partial crash dumps, the size of the files also depends on the level of system activity at the time of the crash. A general guideline is to reserve, at a minimum, the amount of space you estimate you need to save crash dumps, plus 6 MB. The vmunix.*n* file occupies about 6 MB of disk space. You can adjust this amount if need be once your system has attempted to save several crash dump files.

For example, suppose you save partial crash dumps. Your system has 96 MB of memory, but your peak system activity level is 80 MB. You have reserved 85 MB of disk space for crash dumps and swapping. In this case, you should reserve 91 MB of space in the file system for storing crash dump files. You need to reserve considerably more space if you want to save files from more than one crash dump. If you want to save files from multiple crash dumps, consider compressing older crash dump files. See Section 4.6 for information about compressing and uncompressing partial crash dump files.

By default, `savecore` writes crash dump files to the `/var/adm/crash` directory. To reserve space for crash dump files in the default directory, you must mount the `/var/adm/crash` directory on a file system that has a sufficient amount of disk space. (For information about mounting file systems, see the *System Administration* manual and the `mount`(8) reference page.) If you expect your crash dump files to be large, you might need to use a Logical Storage Manager (LSM) file system to store crash dump files. For information about creating LSM file systems, see the *Logical Storage Manager* manual.

If your system cannot save crash dump files due to insufficient disk space, the system returns to single-user mode. This return to single-user mode prevents system swapping from corrupting the crash dump. Once in single-user mode, you can make space available in the crash directory or change the crash directory. One possibility in this situation is to issue the `savecore` command at the single-user mode prompt. On the command line, specify the name of a directory that contains a sufficient amount of file space to save the crash dump files. For example, the following `savecore` command writes crash dump files to the `/usr/adm/crash2` directory:

```
# savecore /usr/adm/crash2
```

Once `savecore` has saved the crash dump files, you can bring your system to multiuser mode.

Specifying a directory on the `savecore` command line changes the crash directory only for the duration of that command. If the system crashes later and the system startup script invokes the `savecore` script, `savecore` copies the crash dump to files in the default directory, which is normally `/var/adm/crash`.

You can control the default location of the crash directory with the `rcmgr` command. For example, to save crash dump files in the `/usr/adm/crash2` directory by default (at each system startup), issue the following command:

```
# /usr/sbin/rcmgr set SAVECORE_DIR /usr/adm/crash2
```

If you want the system to return to multiuser mode, regardless of whether it saved a crash dump, issue the following command:

```
# /usr/sbin/rcmgr set SAVECORE_FLAGS M
```

4.6 Compressing and Uncompressing Crash Dump Files

If you want to store files from more than one crash, you might find it useful to compress the crash dump files. In particular, you should compress the `vmcore.n` files.

If you compress a `vmcore.n` dump file from a partial crash dump, you must use care when you uncompress it. Using the `uncompress` command with no flags results in a `vmcore.n` file requiring space equal to the size of memory. In other words, the uncompressed file requires the same amount of disk space as a `vmcore.n` file from a full crash dump.

This situation occurs because the original `vmcore.n` file contains UNIX File System (UFS) file "holes." UFS files can contain regions, called holes, that have no associated data blocks. When a process, such as the `uncompress` command, reads from a hole in a file, the file system returns zero-valued data. Thus, memory omitted from the partial dump is added back into the uncompressed `vmcore.n` file as disk blocks containing all zeros.

To ensure that the uncompressed core file remains at its partial dump size, you must pipe the output from the `uncompress` command with the `-c` flag to the `dd` command with the `conv=sparse` option. For example, to uncompress a file named `vmcore.0.Z`, issue the following command:

```
# uncompress -c vmcore.0.Z | dd of=vmcore.0 conv=sparse
262144+0 records in
262144+0 records out
```

4.7 Creating Dumps of a Hung System

You can force the system to create a crash dump when the system hangs. On most hardware platforms, you force a crash dump by following these steps:

1. If your system has a switch for enabling and disabling the Halt button, set that switch to the Enable position.
2. Press the Halt button.

3. At the console prompt, enter the `crash` command.

Some systems have no Halt button. In this case, follow these steps to force a crash dump on a hung system:

1. Press Ctrl/P at the console.
2. At the console prompt, enter the `crash` command.

If your system hangs and you force a crash dump, the panic string recorded in the crash dump is the following:

```
hardware restart
```

This panic string is always the one recorded when system operation is interrupted by pressing the Halt button or Ctrl/P.

5

Crash Analysis Examples

Finding problems in crash dump files is a task that takes practice and experience to do well. Exactly how you determine what caused a crash varies depending on how the system crashed. The cause of some crashes is relatively easy to determine, while finding the cause of other crashes is difficult and time-consuming.

This chapter helps you analyze crash dump files by providing the following information:

- Guidelines for examining crash dump files (Section 5.1)
- Examples of identifying the cause of a software panic (Section 5.2)
- Examples of identifying the cause of a hardware trap (Section 5.3)
- An example of finding a panic string that is not in the current thread (Section 5.4)
- An example of identifying the cause of a crash on an SMP system (Section 5.5)

For information about how crash dump files are created, see Chapter 4.

5.1 Guidelines for Examining Crash Dump Files

In examining crash dump files, there is no one way to determine the cause of a system crash. However, following these steps should help you identify the events that lead to most crashes:

1. Gather some facts about the system; for example, operating system type, version number, revision level, hardware configuration.
2. Locate the thread executing at the time of the crash. Most likely, this thread contains the events that lead to the panic.
3. Look at the panic string, if one exists. This string is contained in the preserved message buffer (`pmsgbuf`) and in the `panicstr` global variable. The panic string gives a reason for the crash.

4. Identify the function that called the `panic` or `trap` function. That function is the one that caused the system to crash.

5. Examine the source code for the function that caused the crash to infer the error that caused the crash. You might also need to examine related data structures and functions that appear earlier in the stack. An earlier function might have passed corrupt data to the function that caused a crash.

6. Determine whether you can fix the problem.

   If the system crashed because of a hardware problem (for example, because a memory board became corrupt), correcting the problem probably requires repairing or replacing the hardware. You might be able to disconnect the hardware that caused the problem and operate without it until it is repaired or replaced. If you need to repair or replace Digital hardware, call the nearest Digital service center or sales office.

   If a software panic caused the crash, you can fix the problem if it is in software you or someone else at your company wrote. Otherwise, you must request that the producer of the software fix the problem. If the problem is in software from Digital, you file a Software Performance Report (SPR) to request a correction to the Digital software.

   For information about reporting problems to Digital, contact your local Digital service center or sales office.

5.2 Identifying a Crash Caused by a Software Problem

When software encounters a state from which it cannot continue, it calls the system `panic` function. For example, if the software attempts to access an area of memory that is protected from access, the software might call the `panic` function and crash the system.

In most cases, only system programmers can fix the problem that caused a panic because most panics are caused by software errors. However, some system panics reflect other problems. For example, if a memory board becomes corrupted, software that attempts to write to that board might call the `panic` function and crash the system. In this case, the solution might be to replace the memory board and reboot the system.

The sections that follow demonstrate finding the cause of a software panic using the `dbx` and `kdbx` debuggers. You can also examine output from the crashdc crash data collection tool to help you determine the cause of a crash. Sample output from crashdc is shown and explained in Appendix A.

5.2.1 Using dbx to Determine the Cause of a Software Panic

The following example shows a method for identifying a software panic with the dbx debugger:

```
# dbx -k vmunix.0 vmcore.0
dbx version 3.11.1
Type 'help' for help.

stopped at  [boot:753 ,0xfffffc00003c4ae4]  Source not available

(dbx) p panicstr     1
0xfffffc000044b648 = "ialloc: dup alloc"
(dbx) t              2
>  0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep.\
c":753, 0xfffffc00003c4ae4]
   1 panic(s = 0xfffffc000044b618 = "mode = 0%o, inum = %d, pref = %d fs = %s\n")\
 ["../../../../src/kernel/bsd/subr_prf.c":1119, 0xfffffc00002bdbb0]
   2 ialloc(pip = 0xffffffff8c6acc40, ipref = 57664, mode = 0, ipp = 0xffffffff8c\
f95af8) ["../../../../src/kernel/ufs/ufs_alloc.c":501, 0xfffffc00002dab48]
   3 maknode(vap = 0xffffffff8cf95c50, ndp = 0xffffffff8cf922f8, ipp = 0xffffffff\
8cf95b60) ["../../../../src/kernel/ufs/ufs_vnops.c":2842, 0xfffffc00002ea500]
   4 ufs_create(ndp = 0xffffffff8cf922f8, vap = 0xfffffc00002fe0a0) ["../../../..\
/src/kernel/ufs/ufs_vnops.c":602, 0xfffffc00002e771c]
   5 vn_open(ndp = 0xffffffff8cf95d18, fmode = 4618, cmode = 416) ["../../../../s\
rc/kernel/vfs/vfs_vnops.c":258, 0xfffffc00002fe138]
   6 copen(p = 0xffffffff8c6efba0, args = 0xffffffff8cf95e50, retval = 0xffffffff\
8cf95e40, compat = 0) ["../../../../src/kernel/vfs/vfs_syscalls.c":1379, 0xfffffc\
00002fb890]
   7 open(p = 0xffffffff8cf95e40, args = (nil), retval = 0x7f4) ["../../../../src\
/kernel/vfs/vfs_syscalls.c":1340, 0xfffffc00002fb7bc]
   8 syscall(ep = 0xffffffff8cf95ef8, code = 45) ["../../../../src/kernel/arch/al\
pha/syscall_trap.c":532, 0xfffffc00003cfa34]
   9 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":703, 0xfffffc00003\
c31e0]
(dbx) q
```

1. Display the panic string (panicstr). The panic string shows that the ialloc function called the panic function.
2. Perform a stack trace. This confirms that the ialloc function at line 501 in file ufs_alloc.c called the panic function.

5.2.2 Using kdbx to Determine the Cause of a Software Panic

The following example shows a method of finding a software panic using the kdbx debugger:

```
# kdbx -k vmunix.3 vmcore.3
dbx version 3.11.1
Type 'help' for help.

stopped at  [boot:753 ,0xfffffc00003c4b04]  Source not available

(kdbx) sum        1
Hostname : system.dec.com
cpu: DEC3000 - M500       avail: 1
Boot-time:      Mon Dec 14 12:06:31 1992
Time:   Mon Dec 14 12:17:16 1992
Kernel : OSF1 release 1.2 version 1.2 (alpha)
(kdbx) p panicstr 2
0xfffffc0000453ea0 = "wdir: compact2"
(kdbx) t          3
>  0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep\
.c":753, 0xfffffc00003c4b04]
  1 panic(s = 0xfffffc00002e0938 = "p") ["../../../../src/kernel/bsd/subr_prf.c"\
:1119, 0xfffffc00002bdbb0]
  2 direnter(ip = 0xffffffff00000000, ndp = 0xffffffff9d38db60) ["../../../../sr\
c/kernel/ufs/ufs_lookup.c":986, 0xfffffc00002e2adc]
  3 ufs_mkdir(ndp = 0xffffffff9d38a2f8, vap = 0x100000020) ["../../../../src/ker\
nel/ufs/ufs_vnops.c":2383, 0xfffffc00002e9cbc]
  4 mkdir(p = 0xffffffff9c43d7c0, args = 0xffffffff9d38de50, retval = 0xffffffff\
9d38de40) ["../../../../src/kernel/vfs/vfs_syscalls.c":2579, 0xfffffc00002fd930]
  5 syscall(ep = 0xffffffff9d38def8, code = 136) ["../../../../src/kernel/arch/a\
lpha/syscall_trap.c":532, 0xfffffc00003cfa54]
  6 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":703, 0xfffffc00003\
c3200]
(kdbx) q
dbx (pid 29939) died.  Exiting...
```

1. Use the sum command to get a summary of the system.
2. Display the panic string (panicstr).
3. Perform a stack trace of the current thread block. The stack trace shows that the direnter function, at line 986 in file ufs_lookup.c, called the panic function.

5.3 Identifying a Hardware Exception

Occasionally, your system might crash due to a hardware error. During a hardware exception, the hardware encounters a situation from which it cannot continue. For example, the hardware might detect a parity error in a portion of memory that is necessary for its successful operation. When a hardware exception occurs, the hardware stores information in registers and stops operation. When control returns to the software, it normally calls the panic function and the system crashes.

The sections that follow show how to identify hardware traps using the dbx and kdbx debuggers. You can also examine output from the crashdc crash data collection tool to help you determine the cause of a crash. Sample output from crashdc is shown and explained in Appendix A.

5.3.1 Using dbx to Determine the Cause of a Hardware Error

The following example shows a method for identifying a hardware trap with the dbx debugger:

```
# dbx -k vmunix.1 vmcore.1
dbx version 3.11.1
Type 'help' for help.
(dbx) sh strings vmunix.1 | grep '(Rev'      1
DEC OSF/1 X2.0A-7  (Rev. 1);

(dbx) p utsname        2
struct {
    sysname = "OSF1"
    nodename = "system.dec.com"
    release = "2.0"
    version = "2.0"
    machine = "alpha"
}

(dbx) p panicstr       3
0xfffffc0000489350 = "trap: Kernel mode prot fault\n"

(dbx) t                4
>  0 boot(paniced = 0, arghowto = 0) ["/usr/sde/alpha/build/alpha.nightly/src/ker\
nel/arch/alpha/machdep.c":
    1 panic(s = 0xfffffc0000489350 = "trap: Kernel mode prot fault\n") ["/usr/sde\
/alpha/build/alpha.nightly/src/kernel/bsd/subr_prf.c":1099, 0xfffffc00002c0730]
   2 trap() ["/usr/sde/alpha/build/alpha.nightly/src/kernel/arch/alpha/trap.c":54\
4, 0xfffffc00003e0c78]
   3 _XentMM() ["/usr/sde/alpha/build/alpha.nightly/src/kernel/arch/alpha/locore.\
s":702, 0xfffffc00003d4ff4]

(dbx) kps              5
  PID   COMM
00000   kernel idle
00001   init
00002   device server
00003   exception hdlr
00663   ypbind
00018   cfgmgr
00219   automount
:
:
00265   cron
00293   xdm
02311   inetd
00278   lpd
01443   csh
01442   rlogind
01646   rlogind
01647   csh

(dbx) p $pid           6
```

```
2311

(dbx) p *pmsgbuf       7
struct {
    msg_magic = 405601
    msg_bufx = 62
    msg_bufr = 3825
    msg_bufc = "nknown flag
printstate: unknown flag
printstate: unknown flag
de: table is full
<3>vnode: table is full
:
:
<3>arp: local IP address 0xffffffff82b40429 in use by
hardware address 08:00:2B:20:19:CD
<3>arp: local IP address 0xffffffff82b40429 in use by
hardware address 08:00:2B:2B:F6:3B

va=0000000000000028, status word=0000000000000000, pc=fffffc000032972c
panic: trap: Kernel mode prot fault
syncing disks... 3 3 done
printstate: unknown flag
printstate: unknown flag
printstate: unknown flag
printstate: unknown flag
printstate: u"
}

(dbx) px savedefp
0xffffffff89b2b4e0

(dbx) p savedefp
0xffffffff89b2b4e0

(dbx) p savedefp[28]
18446739675666356012

(dbx) px savedefp[28]       8
0xfffffc000032972c

(dbx) savedefp[28]/i        9
  [nfs_putpage:2344, 0xfffffc000032972c]        ldl     r5, 40(r1)
(dbx) savedefp[23]/i       10
  [ubc_invalidate:1768, 0xfffffc0000315fe0]     stl     r0, 84(sp)

(dbx) func nfs_putpage       11
(dbx) file       12
/usr/sde/alpha/build/alpha.nightly/src/kernel/kern/sched_prim.c
(dbx) func ubc_invalidate        13
```

```
ubc_invalidate:  Source not available

(dbx) file       14
/usr/sde/alpha/build/alpha.nightly/src/kernel/vfs/vfs_ubc.c

(dbx) q
```

1. You can use the sh command to enter commands to the shell. In this case, enter the stings and grep commands to pull the operating system revision number in the vmunix.1 dump file.
2. Display the utsname structure to obtain more information about the operating system version.
3. Display the panic string (panicstr). The panic function was called by a trap function.
4. Perform a stack trace. This confirms that the trap function called the panic function. However, the stack trace does not show what caused the trap.
5. Look to see what processes were running when the system crashed by entering the kps command.
6. Look to see what the process ID (PID) was pointing to at the time of the crash. In this case, the PID was pointing to process 2311, which is the inetd daemon, from the kps command output.
7. Display the preserved message buffer (pmsgbuf). Note that this buffer contains the program counter (pc) value, which is displayed in the following line:

   ```
   va=0000000000000028, status word=0000000000000000, pc=fffffc000032972c
   ```

8. Display register 28 of the exception frame pointer (savedefp). This register always contains the pc value. You can always obtain the pc value from either the preserved message buffer or register 28 of the exception frame pointer.
9. Disassemble the pc to determine its contents. The pc at the time of the crash contained the nfs_putpage function at line 2344.
10. Disassemble the return address to determine its contents. The return value at the time of the crash contained the ubc_invalidate function at line 1768.
11. Point the dbx debugger to the nfs_putpage function.
12. Display the name of the source file that contains the nfs_putpage function.

13. Point the dbx debugger to the ubc_invalidate function.
14. Display the name of the source file that contains the ubc_invalidate function.

The result from this example shows that the ubc_invalidate function, which resides in the /vfs/vfs_ubc.c file at line number 1768, called the nfs_putpage function at line number 2344 in the /kern/sched_prim.c file and the system stopped.

5.3.2 Using kdbx to Determine the Cause of a Hardware Error

The following example shows a method for identifying a hardware error by using the kdbx debugger:

```
# kdbx -k vmunix.5 vmcore.5
dbx version 3.11.1
Type 'help' for help.

stopped at  [boot:753 ,0xfffffc00003c4b04]  Source not available
(kdbx) sum            [1]
Hostname : system.dec.com
cpu: DEC3000 - M500     avail: 1
Boot-time:      Thu Jan  7 08:12:30 1993
Time:   Thu Jan  7 08:13:23 1993
Kernel : OSF1 release 1.2 version 1.2 (alpha)
(kdbx) p panicstr     [2]
0xfffffc0000471030 = "ECC Error"
(kdbx) t              [3]
>  0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep.\
c":753, 0xfffffc00003c4b04]
  1 panic(s = 0x670) ["../../../../src/kernel/bsd/subr_prf.c":1119, 0xfffffc00002\
bdbb0]
  2 kn15aa_machcheck(type = 1648, cmcf = 0xfffffc00000f8050 = , framep = 0xffff\
ffff94f79ef8) ["../../../../src/kernel/arch/alpha/hal/kn15aa.c":1269, 0xfffffc000\
03da62c]
  3 mach_error(type = -1795711240, phys_logout = 0x3, regs = 0x6) ["../../../../s\
rc/kernel/arch/alpha/hal/cpusw.c":323, 0xfffffc00003d7dc0]
  4 _XentInt() ["../../../../src/kernel/arch/alpha/locore.s":609, 0xfffffc00003c3\
148]
(kdbx) q
dbx (pid 337) died.  Exiting...
```

1. Use the sum command to get a summary of the system.
2. Display the panic string (panicstr).
3. Perform a stack trace. Because the kn15aa_machcheck function (which is a hardware checking function) called the panic function, the system crash was probably the result of a hardware error.

5.4 Finding a Panic String in a Thread Other Than the Current Thread

The dbx and kdbx debuggers have the concept of the current thread. In many cases, when you invoke one of the debuggers to analyze a crash dump, the panic string is in the current thread. At times, however, the current thread contains no panic string and so is probably not the thread that caused the crash.

The following example shows a method for stepping through kernel threads to identify the events that lead to the crash:

```
# dbx -k ./vmunix.2 ./vmcore.2
dbx version 3.11.1
Type 'help' for help.
thread 0x8d431c68 stopped at  [thread_block:1305 +0x114,0xfffffc000033961c]   \
 Source not available
(dbx) p panicstr         [1]
0xfffffc000048a0c8 = "kernel memory fault"
(dbx) t                  [2]

>  0 thread_block() ["../../../../src/kernel/kern/sched_prim.c":1305, 0xfffffc0\
e
00033961c]
   1 mpsleep(chan = 0xffffffff8d4ef450 = , pri = 282, wmesg = 0xfffffc000046f\
290 = "network", timo = 0, lockp = (nil), flags = 0) ["../../../../src/kernel/\
bsd/kern_synch.c":267, 0xfffffc00002b772c]
   2 sosleep(so = 0xffffffff8d4ef408, addr = 0xffffffff906cfcf4 = "^P", pri = 2 \
82,tmo = 0) ["../../../../src/kernel/bsd/uipc_socket2.c":612, 0xfffffc00002d3784]
   3 accept1(p = 0xffffffff8f8bfde8, args = 0xffffffff906cfe50, retval = 0xffff \
ffff906cfe40, compat_43 = 1) ["../../../../src/kernel/bsd/uipc_syscalls.c":300 \
, 0xfffffc00002d4c74]
   4 oaccept(p = 0xffffffff8d431c68, args = 0xffffffff906cfe50, retval = 0xffff \
ffff906cfe40) ["../../../../src/kernel/bsd/uipc_syscalls.c":250, 0xfffffc00002d\
4b0c]
   5 syscall(ep = 0xffffffff906cfef8, code = 99, sr = 1) ["../../../../src/kern \
el/arch/alpha/syscall_trap.c":499, 0xfffffc00003ec18c]
   6 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":675, 0xfffffc000\
03df96c]
(dbx) tlist              [3]
thread 0x8d431a60 stopped at   [thread_block:1305 +0x114,0xfffffc000033961c]   \
Source not available
thread 0x8d431858 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
Source not available
thread 0x8d431650 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
Source not available
thread 0x8d431448 stopped at   [thread_block:1305 +0x114,0xfffffc000033961c]   \
Source not available
thread 0x8d431240 stopped at   [thread_block:1305 +0x114,0xfffffc000033961c]   \
Source not available
:
:
thread 0x8d42f5d0 stopped at   [boot:696 ,0xfffffc00003e119c]     Source not
\
available
thread 0x8d42f3c8 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
```

```
Source not available
thread 0x8d42f1c0 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
Source not available
thread 0x8d42efb8 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
Source not available
thread 0x8d42dd70 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]    \
Source not available
(dbx) tset 0x8d42f5d0        4
thread 0x8d42f5d0 stopped at   [boot:696 ,0xfffffc00003e119c]  Source not ava\
ilable
(dbx) t                      5
>  0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/mac\
hdep.c":694, 0xfffffc00003e1198]
  1 panic(s = 0xfffffc000048a098 = "    sp contents at time of fault: 0x%l01\
6x\r\n\n") ["../../../../src/kernel/bsd/subr_prf.c":1110, 0xfffffc00002beef4]
  2 trap() ["../../../../src/kernel/arch/alpha/trap.c":677, 0xfffffc00003ecc70]
  3 _XentMM() ["../../../../src/kernel/arch/alpha/locore.s":828, 0xfffffc000\
03dfb1c]
  4 pmap_release_page(pa = 18446744071785586688) ["../../../../src/kernel/ar\
ch/alpha/pmap.c":640, 0xfffffc00003e3ecc]
  5 put_free_ptepage(page = 5033216) ["../../../../src/kernel/arch/alpha/pma\
p.c" :534, 0xfffffc00003e3ca0]
  6 pmap_destroy(map = 0xffffffff8d5bc428) ["../../../../src/kernel/arch/alp\
ha/p map.c":1891, 0xfffffc00003e6140]
  7 vm_map_deallocate(map = 0xffffffff81930ee0) ["../../../../src/kernel/vm/\
vm_map.c":482, 0xfffffc00003d03c0]
  8 task_deallocate(task = 0xffffffff8d568d48) ["../../../../src/kernel/kern\
/task.c":237, 0xfffffc000033c1dc]
  9 thread_deallocate(thread = 0x4e4360) ["../../../../src/kernel/kern/threa\
d.c":689, 0xfffffc000033d83c]
 10 reaper_thread() ["../../../../src/kernel/kern/thread.c":1952, 0xfffffc00\
0033e920]
 11 reaper_thread() ["../../../../src/kernel/kern/thread.c":1901, 0xfffffc00\
0033e8ac]
(dbx) q
```

1. Display the panic string (panicstr) to view the panic message, if any. This message indicates that a memory fault occurred.
2. Perform a stack trace of the current thread. Because this thread does not show a call to the panic function, you need to look at other threads.
3. Examine the system's threads. The thread most likely to contain the panic is the boot thread because the boot function always executes immediately before the system crashes. If the boot thread does not exist, you must examine every thread of every process in the process list.
4. Point dbx to the boot thread at address 0x8d42f5d0.
5. In this example, the problem is in the pmap_release_page function at line 640 of the pmap.c file.

5.5 Identifying the Cause of a Crash on an SMP System

If you are analyzing crash dump files from an SMP system, you must first determine on which CPU the panic occurred. You can then continue crash dump analysis as you would on a single processor system.

The following example shows a method for determining which CPU caused the crash and which function called the panic function:

```
% dbx -k ./vmunix.1 ./vmcore.1
dbx version 3.11.6
Type 'help' for help.
stopped at  [boot:1494 ,0xfffffc0000442918]  Source not available
(dbx) p ustsname   1
struct {
    sysname = "OSF1"
    nodename = "wasted.zk3.dec.com"
    release = "V3.0"
    version = "358"
    machine = "alpha"
}

(dbx) print paniccpu   2
0
(dbx) p machine_slot[1] 3
struct {
    is_cpu = 1
    cpu_type = 15
    cpu_subtype = 3
    running = 1
    cpu_ticks = {
        [0] 416162
        [1] 83260
        [2] 1401080
        [3] 11821212
        [4] 1095581
    }
    clock_freq = 1024
    error_restart = 0
    cpu_panicstr = 0xfffffc000059f6a0 = "cpu_ip_intr: panic request"
    cpu_panic_thread = 0xffffffff8109a780
}

(dbx) p panicstr   4
0xfffffc0000558ad0 = "simple_lock: uninitialized lock"
(dbx) tset active_threads[paniccpu]   5
stopped at  [boot:1494 ,0xfffffc0000442918]
(dbx) t   6
>  0 boot(0x0, 0x4, 0xac35c0000000a, 0xfffffc00004403fc, 0xfffffc000000000e) \
["../../../../src/kernel/arch/alpha/machdep.c":1494, 0xfffffc0000442918]
   1 panic(s = 0xfffffc0000558b40 = "simple_lock: hierarchy violation") ["../\
   2 simple_lock_fault(slp = 0xfffffc00006292f0, state = 0, caller = 0xfffffc\
000046f384, arg = 0xfffffc0000534fd8 = "session.s_fpgrp_lock", fmt = 0xfffffc\
0000558de8 = "    class already locked: %s\n", error = 0xfffffc0000558b40 = "\
simple_lock: hierarchy violation") ["../../../../src/kernel/kern/lock.c":1558\
, 0xfffffc00003c34ec]
   3 simple_lock_hierarchy_violation(slp = 0xfffffc000046f384, state = 184467\
```

```
39675668500440, caller = 0xfffffc0000558de8, curhier = 5606208) ["../../../..\
/src/kernel/kern/lock.c":1616, 0xfffffc00003c3620]
   4 xnaintr(0xfffffc00005a5158, 0x2, 0xffffffffb53ef238, 0xfffffc000068a754,\
 0xfffffc000055891d) ["../../../../src/kernel/io/dec/netif/if_xna.c":1077, 0x\
fffffc000046f384]
   5 _XentInt(0x2, 0xfffffc0000447174, 0xfffffc00005b7d40, 0x2, 0x0) ["../../\
   6 swap_ipl(0x2, 0xfffffc0000447174, 0xfffffc00005b7d40, 0x2, 0x0) ["../../\
   7 boot(0x0, 0x0, 0xffffffffa52c6000, 0xffffffffb53ef1f8, 0xfffffc00003bf4f\
c) ["../../../../src/kernel/arch/alpha/machdep.c":1434, 0xfffffc000044280c]
   8 panic(s = 0xfffffc0000558ad0 = "simple_lock: uninitialized lock") ["../.\
   9 simple_lock_fault(slp = 0xffffffffa52c6000, state = 1719, caller = 0xfff\
ffc00003734c4, arg = (nil), fmt = (nil), error = 0xfffffc0000558ad0 = "simple\
_lock: uninitialized lock") ["../../../../src/kernel/kern/lock.c":1558, 0xfff\
ffc00003c34ec]
  10 simple_lock_valid_violation(slp = 0xfffffc00003734c4, state = 0, caller \
= (nil)) ["../../../../src/kernel/kern/lock.c":1584, 0xfffffc00003c3578]
  11 pgrp_ref(0xffffffffa52c6000, 0x0, 0xfffffc000023ee20, 0x6b7, 0xfffffc000\
05e1080) ["../../../../src/kernel/bsd/kern_proc.c":561, 0xfffffc00003734c4]
  12 exit(0xffffffffb53ef740, 0x100, 0x1, 0xffffffffa42e5e80, 0x1) ["../../..\
/../src/kernel/bsd/kern_exit.c":868, 0xfffffc000023ef30]
  13 rexit(0xffffffff814d2d80, 0xffffffffb53ef758, 0xffffffffb53ef8b8, 0x1000\
00001, 0x0) ["../../../../src/kernel/bsd/kern_exit.c":546, 0xfffffc000023e7dc]
  14 syscall(0xffffffffb53ec000, 0xfffffc000068a300, 0x0, 0x51, 0x1) ["../../\
  15 _Xsyscall(0x8, 0x3ff800e6938, 0x14000d0f0, 0x1, 0x11ffffc18) ["../../../\
(dbx) p *pmsgbuf  [7]
struct {
    msg_magic = 405601
    msg_bufx = 701
    msg_bufr = 134
    msg_bufc = "0.64.143, errno 22
NFS server: stale file handle fs(742,645286) file 573 gen 32779
 getattr, client address = 16.140.64.143, errno 22

simple_lock: uninitialized lock

    pc of caller:         0xfffffc00003734c4
    lock address:         0xffffffffa52c6000
    lock class name:      (unknown_simple_lock)
    current lock state:   0x00000000e0e9b04a (cpu=0,pc=0xfffffc00e0e9b048,free)

panic (cpu 0): simple_lock: uninitialized lock

simple_lock: hierarchy violation

    pc of caller:         0xfffffc000046f384
    lock address:         0xfffffc00006292f0
    lock info addr:       0xfffffc0000672cc0
    lock class name:      xna_softc.lk_xna_softc
    class already locked: session.s_fpgrp_lock
:
:
}
(dbx) quit
```

1. Display the `ustname` structure to obtain information about the system.

2. Display the number of the CPU on which the panic occurred, in this case CPU 0 was the CPU that started the system panic.
3. Display the machine_slot structure for a CPU other than the one that started the system panic. Notice that the panic string contains:

   ```
   cpu_ip_intro: panic_request
   ```

   This panic string indicates that this CPU was not the one that started the system panic. This CPU was requested to panic and stop operation.
4. Display the panic string, which in this case indicates that a process attempted to obtain an uninitialized lock.
5. Set the context to the CPU that caused the system panic to begin.
6. Perform a stack trace on the CPU that started the system panic.

   Notice that the panic function appears twice in the stack trace. The series of events that resulted in the first call to the panic function caused the crash. The events that occurred after the first call to the panic function were performed after the system was corrupt and during an attempt to save data. Normally, any events that occur after the initial call to the panic function will not help you determine why the system crashed.

   In this example, the problem is in the pgrp_ref function on line 561 in the kern_proc.c file.

   If you follow the stack trace after the pgrp_ref function, you can see that the pgrp_ref function calls the simple_lock_valid_violation function. This function displays information about simple locks, which might be helpful in determining why the system crashed.
7. Retrieve the information from the simple_lock_valid_violation function by displaying the preserved message buffer.

A

Output from the crashdc Command

This appendix contains a sample `crash-data.n` file created by the `crashdc` command. The output is explained in the list following the example.

```
#
# Crash Data Collection (Version 1.4)
#
_crash_data_collection_time: Fri Sep  2 15:01:07 EDT 1994   1
_current_directory: /
_crash_kernel: /var/adm/crash/vmunix.0
_crash_core: /var/adm/crash/vmcore.0
_crash_arch: alpha
_crash_os: DEC OSF/1
_host_version: DEC OSF/1 3.0  (Rev. 331); Thu Sep  1 09:24:01 EDT 1994
_crash_version: DEC OSF/1 3.0  (Rev. 331); Thu Sep  1 09:24:01 EDT 1994
_crashtime:  struct {
    tv_sec = 746996332
    tv_usec = 145424
}
_boottime:  struct {
    tv_sec = 746993148
    tv_usec = 92720
}
_config:  struct {
    sysname = "OSF1"
    nodename = "madmax.zk3.dec.com"
    release = "3.0"
    version = "331"
    machine = "alpha"
}
_cpu:  30
_system_string:  0xfffffc0000442fa8 = "DEC3000 - M500"
_avail_cpus:  1
_partial_dump:  1
_physmem(MBytes):  96
_panic_string:  0xfffffc000043cf70 = "kernel memory fault" 2
_preserved_message_buffer_begin: 3
struct {
    msg_magic = 0x63061
    msg_bufx = 0x56e
    msg_bufr = 0x432
    msg_bufc = "Alpha boot: available memory from 0x678000 to 0x6000000
DEC OSF/1 T2.0-1  (Rev. 114.2); Wed Sep  1 09:24:01 EDT 1993
physical memory = 94.00 megabytes.
available memory = 84.50 megabytes.
using 360 buffers containing 2.81 megabytes of memory
tc0 at nexus
scc0 at tc0 slot 7
```

```
tcds0 at tc0 slot 6
asc0 at tcds0 slot 0
rz0 at asc0 bus 0 target 0 lun 0 (DEC     RZ26     (C) DEC T384)
rz4 at asc0 bus 0 target 4 lun 0 (DEC     RRD42   (C) DEC  4.5d)
tz5 at asc0 bus 0 target 5 lun 0 (DEC     TLZ06     (C)DEC 0374)
asc1 at tcds0 slot 1
rz8 at asc1 bus 1 target 0 lun 0 (DEC     RZ57     (C) DEC 5000)
rz9 at asc1 bus 1 target 1 lun 0 (DEC     RZ56     (C) DEC 0300)
fb0 at tc0 slot 8
 1280X1024
bba0 at tc0 slot 7
ln0: DEC LANCE Module Name: PMAD-BA
ln0 at tc0 slot 7
ln0: DEC LANCE Ethernet Interface, hardware address: 08-00-2b-2c-f3-83
DEC3000 - M500 system
Firmware revision: 2.4
PALcode: OSF version 1.28
lvm0: configured.
lvm1: configured.
<3>/var: file system full
<3>/var: file system full
<3>/var: file system full
<3>/var: file system full
<3>/var: file system full

trap: invalid memory ifetch access from kernel mode

    faulting virtual address:     0x0000000000000000
    pc of faulting instruction:   0x0000000000000000
    ra contents at time of fault: 0xfffffc000028951c
    sp contents at time of fault: 0xffffffff96199a48

panic: kernel memory fault
syncing disks... done
"
}
_preserved_message_buffer_end:
_kernel_process_status_begin:  [4]
  PID COMM
00000 kernel idle
00001 init
00002 exception hdlr
00342 xdm
00012 update
00341 Xdec
00239 nfsiod
00113 syslogd
00115 binlogd
00240 nfsiod
00241 nfsiod
00340 csh
00124 routed
00188 portmap
00197 ypbind
00237 nfsiod
00249 sendmail
00294 internet_mom
00297 snmp_pe
00291 mold
00337 xdm
```

```
00325 lpd
00310 cron
00305 inetd
00489 tar
_kernel_process_status_end:
_current_pid:  489  [5]
_current_tid:  0xffffffff863d36c0  [6]
_proc_thread_list_begin:
thread 0x863d36c0 stopped at [boot:1118,0xfffffc0000374a08] Source not available
_proc_thread_list_end:
_dump_begin: [7]
>  0 boot(reason = 0, howto = 0) ["../../../../src/kernel/arch/alpha/machdep.c":
1118, 0xfffffc0000374a08]
mp = 0xffffffff961962f8
nmp = 0xffffffff86333ab8
fsp = (nil)
rs = 5368785696
error = -1776721160
ind = 2424676
nbusy = 4643880

   1 panic(s = 0xfffffc000043cf70 = "kernel memory fault") ["../../../../src\
/kernel/bsd/subr_prf.c"\
:616, 0xfffffc000024ff60]
bootopt = 0

   2 trap() ["../../../../src/kernel/arch/alpha/trap.c":945, 0xfffffc0000381440]
t = 0xffffffff863d36c0
pcb = 0xffffffff96196000
task = 0xffffffff86306b80
p = 0xffffffff95aaf6a0
syst = struct {
    tv_sec = 0
    tv_usec = 0
}
nofault_save = 0
exc_type = 18446739675665756628
exc_code = 0
exc_subcode = 0
i = -2042898428
s = 2682484
ret = 536993792
map = 0xffffffff808fc5a0
prot = 5
cp = 0xffffffff95a607a0 =
i = 0
result = 18446744071932830456
pexcsum = 0xffffffff00000000
i = 16877
pexcsum = 0xffffffff00001000
i = 2682240
ticks = -1784281184
tv = 0xfffffffc00500068

   3 _XentMM() ["../../../../src/kernel/arch/alpha/locore.s":949, 0xfffff\
c0000372dec]

_dump_end:

warning: Files compiled -g3: parameter values probably wrong
```

```
_kernel_thread_list_begin: 8
thread 0x8632faf0 stopped at [thread_block:1427 ,0xfffffc00002ca3a0]  Source\
 not available
thread 0x8632f8d8 stopped at [thread_block:1427 ,0xfffffc00002ca3a0]  Source\
 not available
:
thread 0x8632d328 stopped at [thread_block:1400 +0x1c,0xfffffc00002ca2f8]  \
Source not available
thread 0x8632d110 stopped at [thread_block:1400 +0x1c,0xfffffc00002ca2f8]  \
Source not available
_kernel_thread_list_end:
_savedefp:  0xffffffff96199940 9
_kernel_memory_fault_data_begin:  10
struct {
    fault_va = 0x0
    fault_pc = 0x0
    fault_ra = 0xfffffc000028951c
    fault_sp = 0xffffffff96199a48
    access = 0xffffffffffffffff
    status = 0x0
    cpunum = 0x0
    count = 0x1
    pcb = 0xffffffff96196000
    thread = 0xffffffff863d36c0
    task = 0xffffffff86306b80
    proc = 0xffffffff95aaf6a0
}
_kernel_memory_fault_data_end:
Invalid character in input
_uptime: .88 hours

_stack_trace_begin: 11
>  0 boot(reason = 0, howto = 0) ["../../../../src/kernel/arch/alpha/machdep.c"\
:1118, 0xfffffc0000374a08]
   1 panic(s = 0xfffffc000043cf70 = "kernel memory fault") ["../../../. ./src\
/kernel/bsd/subr_prf.c":616, 0xfffffc000024ff60]
   2 trap() ["../../../../src/kernel/arch/alpha/trap.c":945, 0xfffffc0000381\
440]
   3 _XentMM() ["../../../../src/kernel/arch/alpha/locore.s":949, 0xfffffc000\
0372dec]
_stack_trace_end:
_savedefp_exception_frame_(savedefp/33X):  12
ffffffff96199940:  0000000000000000 fffffc000046f888
ffffffff96199950:  ffffffff863d36c0 0000000079c2c93f
ffffffff96199960:  000000000000007d 0000000000000001
ffffffff96199970:  0000000000000000 fffffc000046f4e0
ffffffff96199980:  0000000000000000 ffffffff961962f8
ffffffff96199990:  0000000140012b20 0000000000000000
ffffffff961999a0:  0000000140045690 0000000000000000
ffffffff961999b0:  00000001400075e8 0000000140026240
ffffffff961999c0:  ffffffff96199af0 ffffffff8635adc0
ffffffff961999d0:  ffffffff96199ac0 00000000000001b0
ffffffff961999e0:  fffffc00004941b8 0000000000000000
ffffffff961999f0:  0000000000000001 fffffc000028951c
ffffffff96199a00:  0000000000000000 0000000000000fff
ffffffff96199a10:  0000000140026240 0000000000000000
ffffffff96199a20:  0000000000000000 fffffc000047acd0
:
```

```
ffffffff96199a30:  0000000000901402 0000000000001001
ffffffff96199a40:  0000000000002000
_savedefp_exception_frame_ptr:  0xffffffff96199940
_savedefp_stack_pointer:  0x140026240
_savedefp_processor_status:  0x0
_savedefp_return_address:  0xfffffc000028951c
_savedefp_pc:  0x0
_savedefp_pc/i:

can't read from process (address 0x0)
_savedefp_return_address/i:
  [spec_open:997, 0xfffffc000028951c]  bis r0, r0, r19
_kernel_memory_fault_data.fault_pc/i:

can't read from process (address 0x0)
_kernel_memory_fault_data.fault_ra/i:
  [spec_open:997, 0xfffffc000028951c]  bis r0, r0, r19

_kdbx_sum_start:
Hostname : madmax.zk3.dec.com
cpu: DEC3000 - M500     avail: 1
Boot-time:      Thu Sep  2 14:05:48 1993
Time:   Thu Sep  2 14:58:52 1993
Kernel : OSF1 release T2.0 version 114.2 (alpha)
_kdbx_sum_end:
_kdbx_swap_start:  [13]
       Swap device name           Size       In Use       Free
-----------------------------  ----------  ----------  ----------
/dev/rz0b                       131072k      10560k     120512k  Dumpdev
                                 16384p       1320p      15064p
-----------------------------  ----------  ----------  ----------
Total swap partitions:  1       131072k      10560k     120512k
                                 16384p       1320p      15064p
_kdbx_swap_end:
_kdbx_proc_start:  [14]
Addr         PID  PPID  PGRP  UID  NICE  SIGCATCH P_SIG    Event  Flags
===========  ===  ====  ====  ===  ====  ======== ======== =====  =====
v0x95aaf6a0  489  340   489    0     0   00000000 00000000 NULL   in pagv ctty
v0x95aad5d0  342  337   342    0     0   00000000 00000000 NULL   in pagv ctty
v0x95aad8f0  341  337   341    0     0   00000000 00000000 NULL   in pagv
 .
 .
 .
v0x95aad2b0    1    0     1    0     0   00000000 00000000 NULL   in omask pagv
v0x95aad120    0    0     0    0     0   00000000 00000000 NULL   in sys
kdbx_proc_end:

Audit subsystem not installed
#
_crash_data_collection_finished:
```

1. The first several lines of output display the contents of system variables that give statistics about the crash, such as:

   - The kernel image file and crash core file from which `crashdc` collected data.
   - The operating system version.

- The time of the crash and the time at which the system was rebooted.
- Whether data is from a partial or full dump. (Data is from a partial dump when the value of the `partial_dump` variable is 1. Data is from a full dump when the value of this variable is 0.)
- The platform on which the operating system is running; a DECstation 3000 in this case.
- The amount of physical memory available on the system.

2. The `_panic_string` label marks the message that indicates why the crash occurred. In this case the message is `kernel memory fault`, indicating that a memory operation failed in the kernel.
3. The preserved message buffer contains status and other information about the devices connected to the system: Notice the following message:

   ```
   trap:  invalid memory ifetch access from kernel mode
   ```

   This message describes the kernel memory fault and indicates that the kernel was unable to fetch a needed instruction.

   The preserved message buffer also contains the faulting virtual address, the pc of the instruction that failed, the contents of the return address register, and the stack pointer at the time of the memory fault.
4. The kernel process status list shows the processes that were active at the time of the crash.
5. The `_current_pid` label marks the process ID of the process that was executing at the time of the crash. In this case, it is the `tar` process, which is identified as process 489 in the kernel process status list.
6. The `_current_tid` label marks the address of the thread that was executing at the time of the crash.
7. The dump section shows information about the variables passed to the routines executing at the time of the crash. In this case, the dump displays variable information for the `boot`, `panic`, and `trap` functions.
8. The kernel thread list shows the threads of execution in the kernel. This information can be helpful for verifying which routine called the `panic` function.
9. The `savedefp` variable contains a pointer to the exception frame.

10. The kernel memory fault data displays the following information, recorded at the time of the memory fault:

    - The fault_va variable contains the faulting virtual address.
    - The fault_pc variable contains the pc.
    - The fault_ra variable contains the return address of the calling routine.
    - The fault_sp variable contains the stack pointer.
    - The access variable contains the access code, which is zero (0) for read access, 1 for write access, and -1 for execute access.
    - The status variable contains the process status register.
    - The cpunum variable contains the number of the CPU that faulted.
    - The count variable contains the number of CPUs on the system.
    - The pcb variable contains a pointer to the process control block.
    - The thread variable contains a pointer to the current thread.
    - The task variable contains a pointer to the current task.
    - The proc variable contains the address of the process status table.

11. The _stack_trace_begin line begins a trace of the current thread block's stack at the time of the crash. In this case the _XentMM function called the trap function. The trap function called the panic function, which called the boot function and the system crashed.

12. The exception frame is a stack frame created to store the state of the process running at the time of the exception. It stores the registers and pc associated with the process. To determine where registers are stored in the exception frame, refer to the /usr/include/machine/reg.h header file.

13. Swap information is shown to help you determine whether swap space is sufficient.

14. The process table gives information about the processes active at the time of the crash. The information includes:

    - The process ID of each process.
    - The process ID of the parent process for each process.
    - The process group ID for each process.

- The UID of the of the user that started each process. In this case all process are started by `root`.
- The priority at which the process was running at the time of the memory fault.
- The event the process was waiting for, if any. An event might be the completion of an input or output request, for example.
- Any flags assigned to the process. For example, the `ctty` flag indicates that the process has a controlling terminal and, the `sys` flag indicates that the process is a swapper or pager process.

Index

A

B

C

D

L

M

N

O

P

Q

R

S

T

U

V

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 *or* U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader's Comments

Digital UNIX
Kernel Debugging
AA-PS2TE-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 881-0120, Attn: UEG Publications, ZK03-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Usability (ability to access information quickly)	☐	☐	☐	☐

Please list errors you have found in this manual:

Page Description

_________ ____
_________ ____
_________ ____
_________ ____

Additional comments or suggestions to improve this manual:

____
____
____
____

What version of the software described by this manual are you using? ____________

Name, title, department ____________________________________
Mailing address ____________________________________
Electronic mail ____________________________________
Telephone ____________________________________
Date ____________________________________

Do Not Cut or Tear − Fold Here and Tape

digital™

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on Dotted Line