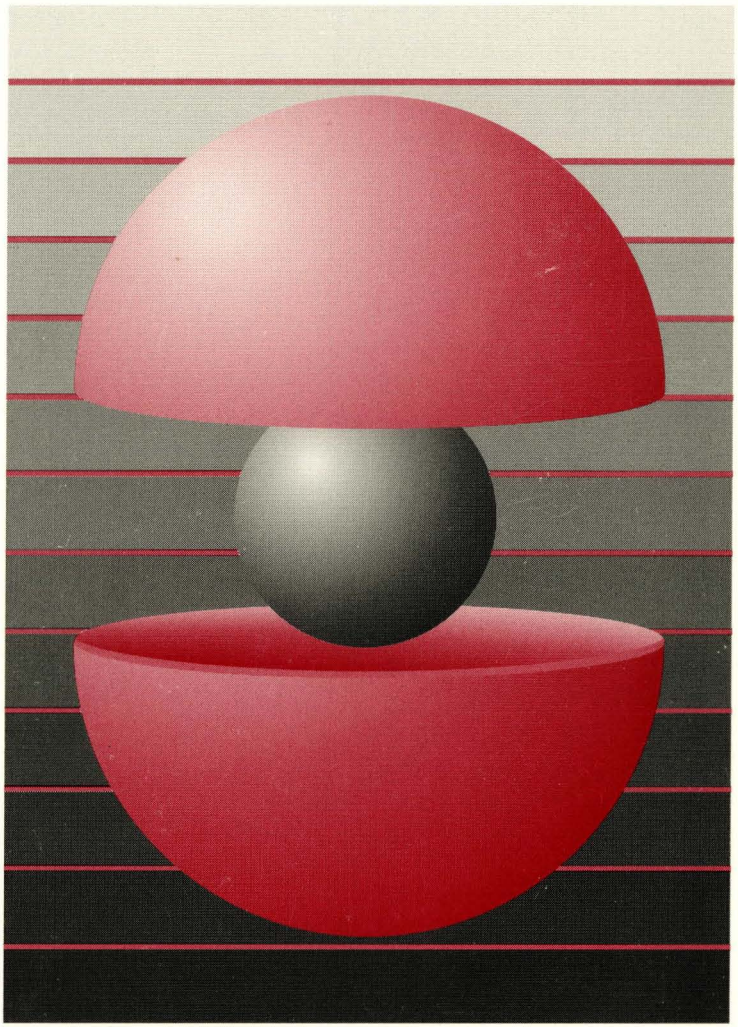
DEC OSF/1

digital

Kernel Debugging



Part Number: AA-PS2TB-TE

DEC OSF/1

Kernel Debugging

Order Number: AA-PS2TB-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This manual explains how to use the tools to debug a kernel and analyze a crash dump of the DEC OSF/1 operating system. Also, this manual explains how to write extensions to the kernel debugging tools.

digital equipment corporation
Maynard, Massachusetts

Contents

About This Manual

1 Kernel Debugging and System Crash Analysis Overview

2 Kernel Debugging Utilities

5 Using the System Exercisers

Index

Examples

Tables

About This Manual

This manual provides information on the tools used to debug a kernel and analyze a crash dump of the DEC OSF/1 operating system. It also explains how to write extensions to the kernel debugging tools. You can use extensions to pull customized information from a kernel or crash dump.

Audience

This manual is intended for system administrators who are responsible for managing the operating system, and for systems programmers who are writing applications for the operating system. Administrators and programmers should have in-depth knowledge of the operating system concepts, commands, and utilities, especially the `dbx` debugger.

This manual assumes that the operating system has been installed.

Organization

This manual consists of 5 chapters:

Chapter 1	Provides an overview of kernel debugging and crash dump analysis.
Chapter 2	Describes the tools used to analyze kernels and crash dump files.
Chapter 3	Provides examples with commentary to show how to analyze a running kernel or crash dump.
Chapter 4	Describes how to write a `kdbx` debugger extension. This chapter assumes you have access to source files (for example, have purchased and installed a DEC OSF/1 Source Kit).
Chapter 5	Describes the various exercise utilities for disk, tape, memory, and communications devices.

Related Documents

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 DEC OSF/1 operating system.

The printed version of the DEC OSF/1 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	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

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* provides information on all of the books in the DEC OSF/1 documentation set.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail:
  `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed DEC OSF/1 manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

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).

Kernel Debugging and System Crash Analysis Overview 1

This chapter contains an overview of kernel debugging and system crash analysis. It also includes information pertaining to the DEC OSF/1 operating system.

1.1 Background Knowledge Needed for Kernel Debugging

Tracing a problem can require a basic understanding of one or more of the following technical areas: the hardware architecture, the internal design of the operating system at a source code and data structure level, and the compilers and programming languages used to create the operating system.

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

See the *Alpha Architecture Reference Manual* for information on how the OpenVMS and DEC OSF/1 operating systems interface with the hardware.

1.2 System Crash and Recovery Process

If the system panics due to an unrecoverable software state or hardware fault, a dump function is invoked. The dump function copies the core memory into the primary default swap disk area as specified by the `/etc/fstab` file structure table and the `/sbin/swapdefault` file.

You can also invoke the dump function at the console prompt. This ability is especially valuable because it enables you to force crashes on hung systems.

The following sections describe procedures for obtaining and creating dump files prior to any debugging or analysis of these files.

1.2.1 Saving Dumps to a File System

When the system reboots, it attempts to save a crash dump from the crash dump partition to a file system. The `savecore` utility (`/sbin/savecore`), which is invoked during system startup before the dump partition is accessed, checks to see if the system crashed or was simply rebooted. If the system crashed recently, the `savecore` utility performs the

following tasks as the system reboots:

- Checks to see if a dump has been made recently (within the last three days) and that there is enough space to save it.
- Saves the dump file and kernel image into a specified directory. The default files for the kernel image and the dump file are `vmunix.n` and `vmcore.n`, respectively.
- Logs a reboot message using the facility `LOG_CRIT`, which logs critical conditions. For more information, refer to the `syslog`(3) reference page.
- Logs the panic string in both the ASCII and binary error log files, if the system crashed as a result of a panic.
- Attempts to save the kernel `syslog` message buffer from the dump files. The `msgbuf.err` entry in `/etc/syslog.conf` file specifies the file name and location for the `msgbuf` dump file. The default `/etc/syslog.conf` file specification is as follows:

  ```
  msgbuf.err                /var/adm/crash/msgbuf.savecore
  ```

  If the `msgbuf.err` entry is not specified in the `/etc/syslog.conf` file, the `msgbuf` dump file is not saved. The `msgbuf` dump file cannot be forwarded to any system.

  When the `syslogd` daemon is later initialized, it checks for the `msgbuf` dump file. If a `msgbuf` dump file is found, `syslogd` processes the file and then deletes it.
- Creates the file `binlogdumpfile.n` in the `/var/adm/crash` directory. The variable *n* is determined by the value of the `bounds` file.

You can modify the system default for the location of dump files by using the `rcmgr` command to specify another directory path for the `/sbin/savecore` utility:

```
# /usr/sbin/rcmgr set SAVECORE_DIR </newpath>
```

The `/sbin/init.d/savecore` script invokes the `/sbin/savecore` utility.

1.2.2 Crash Dump Files

Crash dump files are either partial (the default) or full. The following sections describe each type and present guidelines for allocating the proper amount of space in the crash dump partition and file system.

1.2.2.1 Partial Crash Dump Files

Unlike full crash dumps, the size of a partial crash dump file is proportional to the amount of system activity at the time of the crash. That is, the higher the level of system activity and the larger amount of memory in use at the time of a crash, the larger the partial crash dump files will be. For example, when a system with 96 megabytes of memory crashes, it creates a `vmcore.n` file with 10-to-96 megabytes of memory (depending upon system activity) and a `vmunix.n` file with approximately six megabytes of memory.

Note

If you compress a core dump file from a partial crash dump, you must use care in decompressing it. Using the `uncompress` command with no options results in a core file equal to the size of memory. To ensure that the decompressed core file remains at its partial dump size, you need to use the `uncompress` command with the `-c` option and the `dd` command with the `conv=sparse` option. For example, to decompress a core file named `vmunix.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
```

1.2.2.2 Full Crash Dumps

Full crash dump files can be very large because `vmunix.n` is a copy of the running kernel and the size of `vmcore.n` is slightly larger than the amount of physical memory on the system that crashed. For example, when a system with 96 megabytes of memory crashes, it creates a `vmcore.n` file with approximately 96 megabytes of memory and a `vmunix.n` file with approximately six megabytes of memory.

1.2.2.3 Selecting a Crash Dump Type

The default is to use partial crash dumps. If you want to use full dumps, you can modify the default behavior in the following ways:

- By specifying the `d` flag to the `boot_osflags` console environment variable.
- By modifying the kernel's `partial_dump` variable to 0 using the `dbx`

debugger (discussed in Chapter 2) as follows:

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

Note that a `partial_dump` value of 1 indicates that partial dumps are to be generated.

1.2.2.4 Determining Crash Dump Partition Size

If you intend to save full crash dumps, you need to reserve disk space equal to the size of memory, plus one additional block for the dump header. For example, if your system has 128 megabytes of memory, you need a crash dump partition of at least 128 megabytes, plus one block (512 bytes).

If you intend to save partial crash dumps, the size of the disk partition may vary, depending upon system activity. For example, for a system with 128 megabytes of memory, if peak system activity is low (never using more than 60 megabytes of memory), the size of the crash dump partition can be 60 megabytes. If peak system activity is high (using all of memory), 128 megabytes of disk space is needed.

If full dumps are turned on and there is not enough disk space to create dump files for a full dump, partial dumps are automatically invoked.

1.2.2.5 Determining File System Space for Saving Crash Dumps

The size of the file system needed for saving crash dumps depends on the size and the number of crash dumps you want to retain. A general guideline is to reserve, at a minimum, the size of your crash dump partition, plus 10 megabytes. If necessary, you can increase this amount later.

If your system cannot save a crash dump due to insufficient disk space, it returns to single user mode. This is done to prevent system swapping from corrupting the dump. Space can then be made available in the crash dump directory, or the directory changed as described in Section 1.2.1, before continuing to multiuser mode. This option can also be overidden using the following command:

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

This command causes the system to always boot to multiuser mode even if it cannot save a dump.

1.2.3 The bounds File

The `bounds` file is created in the crash dump directory after the first crash and contains the integer value used for the version number (*n*) for the `vmunix.`*n* and `vmcore.`*n* files. The integer value is incremented for the first crash and every crash thereafter.

1.3 Types of System Failures

The operating system can crash in at least four distinct ways:

- Hardware trap
- Software panic
- Hung system
- Resource exhaustion

Sometimes these crashes are intermittent, and sometimes they are fairly easy to reproduce. The following sections describe each type of failure.

1.3.1 Hardware Trap

A hardware problem often results in the kernel `trap()` function being invoked. In this case, certain variables, such as `savedefp`, are set for later use when diagnosing dump files. If an exception occurs, the trap routine variables, such as the program counter (`pc`) and the stack pointer (`sp`), are readily obtainable for later debugging references.

1.3.2 Software Panic

A software panic, resulting from a software failure, calls the kernel `panic()` function. The function that called the panic routine usually can be identified by looking at the crash dump using the `dbx` debugger or by examining the error logging file with the `uerf` utility. After identifying the function, you need to examine the source code to determine why it failed.

1.3.3 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 1.4 describes the procedure for forcing a crash dump of a hung system.

1.3.4 Resource Exhaustion

In some instances, the system tries to utilize more resources (for example, swap space or memory) than the system has available. In some instances, the only way to fix this type of problem is to add more resources, such as swap space or memory.

1.4 Procedures for Creating Dumps of a Hung System

If necessary, you can force the system to create dump files when the system hangs. The method for forcing crash dumps varies between the various hardware platforms:

- DEC 3000-series systems

  1. Press the reset button.
  2. At the console prompt, issue the `c` command as follows:

     ```
     >>> c
     ```

- DEC 4000 systems

  1. Press the Halt button.
  2. At the console prompt, issue the `crash` command as follows:

     ```
     >>> crash
     ```

- DEC 7000 systems

  1. Ensure that the front panel switch is set to Enable.
  2. Type Ctrl/P at the console.
  3. At the console prompt, issue the `crash` command as follows:

     ```
     >>> crash
     ```

If none of the above methods work, you can force the crash dump using the following method:

1. Find the address of `start` (the function that initializes the system at boot time) by using the `nm` and `grep` utilities as follows:

   ```
   # nm /vmunix | grep '^start '
   start    |fffffc0000254004|Proc   |ref=5     |   |88986|Text
      .
      .
      .
   ```

2. When the system hangs, press the reset button.
3. Set the radix to hexadecimal as follows:

   ```
   >>>set radix 16
   ```

4. Force the system to dump a copy of core memory by loading the address of `start` (plus 4) at the console prompt. For example, if the the address of `start` is `fffffc0000254004`, do the following for DEC 3000-series systems:

   ```
   >>> start fffffc0000254008
   ```

   For DEC 4000 and DEC 7000 systems, do the following:

   ```
   >>> deposit pc fffffc0000254008
   >>> continue
   ```

   Do not precede the address with "0x" when typing the `start` address.

Kernel Debugging Utilities 2

This chapter discusses the various debuggers available to debug kernels and analyze crash dumps:

- `dbx` (Section 2.1)
- `kdbx` (Section 2.2)
- `kdebug` (Section 2.3)

It also describes the `crashdc` utility (Section 2.4), which automatically collects system information from crash dumps.

2.1 The dbx Debugger

The `dbx` debugger is a symbolic debugger that is capable of examining, modifying, and displaying the variables and data structures found in the nonstripped kernel images.

By default, the kernel is compiled with a debugging option that does not strip all of the symbol table information for 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.

The default C compiler for the DEC OSF/1 operating system produces an output file in common output file format (COFF). The `dbx` debugger is able to use COFF files that are nonstripped and either nonoptimized or partially optimized. Output file formats produced by other C compilers, such as `ROSE` or `ELF`, can neither be interpreted by this version of `dbx` nor supported by the DEC OSF/1 operating system.

The following sections describe the `dbx` debugger as it pertains to kernel debugging. For more information on `dbx`, see the *Programmer's Guide*.

2.1.1 Kernel Debugging Option

The `dbx` option `-k` 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 may need to be logged in as `root` to examine either the running system or crash dumps produced by `savecore`. This depends on the directory and file protections for the files you are attempting to examine with the `dbx` debugger.

Use the following command to examine the running system with `dbx`:

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

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

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

2.1.2 Extracting Information from Kernel Images

You can extract information from kernel images with `dbx`. 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: 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.

Table 2-1: (continued)

Mode	Description
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
(dbx) cpup.system_string/s
fffffc00004660d8: "DEC3000 - M500"
```

2.1.2.1 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 = "decosf.dec.com"
    release = "1.4"
    version = "1.2"
    machine = "alpha"
}
```

Note that dbx has a default alias of p for print.

```
(dbx) p utsname
```

2.1.2.2 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;
};
```

2.1.3 Multithreaded Debugging

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

`p $tid`
: Show the thread ID of the current thread.

`p $pid`
: Show the process ID of the current process.

`t`
: Show a stack trace for the current thread.

`tlist`
: Show a list of kernel threads for the current process.

`kps`
: Show 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.

2.1.4 Exception Frame Examination

The `dbx` debugger can be used to examine the exception frame as an aid in debugging crash dumps. The variable `savedefp` contains the location of the exception frame. (Note that exception frames are not created when systems are forced to dump, as described in Section 1.4.) Refer to the header file `/usr/include/machine/reg.h` to determine where registers are stored in the exception frame. The following example shows an exception frame:

```
(dbx) p savedefp/33X
ffffffff9618d940:  0000000000000000 fffffc000046f888
ffffffff9618d950:  ffffffff86329ed0 0000000079cd612f
   •
   •
   •
ffffffff9618da30:  0000000000901402 0000000000001001
ffffffff9618da40:  0000000000002000
```

2.1.5 Character Message Buffer

The preserved message buffer can be extracted from the running system and dump files to display system messages logged by the kernel. For example:

```
(dbx) p *pmsgbuf
struct {
    msg_magic = 405601
    msg_bufx = 1181
    msg_bufr = 1181
    msg_bufc = "Alpha boot: memory from 0x68a000 to 0x6000000
DEC OSF/1 T1.2-2  (Rev. 5); Thu Dec 03 11:20:36 EST 1992
physical memory = 94.00 megabytes.
available memory = 83.63 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 asc0 bus 0 target 1 lun 0 (DEC     RZ25     (C) DEC 0700)
rz2 at asc0 bus 0 target 2 lun 0 (DEC     RZ25     (C) DEC 0700)
rz3 at asc0 bus 0 target 3 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 tc0 slot 6
fb0 at tc0 slot 8
 1280X1024
ln0: DEC LANCE Module Name: PMAD-BA
ln0 at tc0 slot 7
ln0: DEC LANCE Ethernet Interface, hardware address: 08:00:2b:2c:f6:9f
DEC3000 - M500 system
Firmware revision: 1.1
PALcode: OSF version 1.14
```

```
lvm0: configured.
lvm1: configured.
setconf: bootdevice_parser translated 'SCSI 0 6 0 0 300 0 FLAMG-IO' to 'rz3'
"
}
(dbx)
```

2.2 The kdbx Debugger

The kdbx debugger is an interactive program that enables you to examine either the running kernel or dump files created by the savecore utility. In either case, you will be examining an object file and a core file. For running systems, these are usually /vmunix and /dev/mem, respectively. Dump files created by savecore are saved in the directory specified by the /sbin/init.d/savecore script. By default, the savecore script directs dump files to be saved in the /var/adm/crash directory.

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 sizes of fields 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 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 through kdbx using the dbx option to kdbx.

2.2.1 Beginning a kdbx Session

If you do not specify a core file, kdbx uses the dbx default of /dev/mem. Therefore, you can use kdbx with /vmunix as its only argument to examine a running system. In general, kdbx assumes hexadecimal addresses for commands that perform I/O.

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 that are automatically executed when you begin a kdbx session. (The aliases defined in the system.kdbxrc file are listed in Section 2.2.4.) You can further customize the kdbx environment by adding commands and aliases to one of the following initialization files:

- `/var/kdbx/site.kdbxrc`

  Contains customized commands and alias definitions for a particular system.

- `~/.kdbxrc`

  Contains customized commands and alias definitions for a specific user.

- `./.kdbxrc`

  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 Examining Running Systems and Dump Files

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

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

stopped at  [thread_block:1403 ,0xfffffc000032e3c0]    Source not available
(kdbx)
```

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

```
# kdbx -k /usr/adm/crash/vmunix.1 /usr/adm/crash/vmcore.1
dbx version 3.12.1
Type 'help' for help.

stopped at  [thread_block:1403 ,0xfffffc000032e3c0]    Source not available
(kdbx)
```

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

2.2.3 kdbx Debugger Commands

The `kdbx` debugger provides the following commands:

`alias [` *name* `] [` *command-string* `]`
: Sets or prints aliases. If no arguments are specified, `alias` prints all aliases. If the variable *name* is specified, `alias` prints the alias for *name*, if one exists. If *name* and *command-string* are specified, `alias` establishes *name* as an alias for *command-string*.

`context proc | user`
: Sets context to the user's aliases or the extension's aliases. 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 variable *command-string* to `dbx`. Specifying `dbx` is optional; if the command is not recognized by `kdbx`, it is passed to `dbx` automatically. See the `dbx`(1) reference page for a complete description of `dbx` commands.

`help [-long ] [`*args* `]`
Prints help text.

`proc [`*options* `] [`*extension* `] [`*arguments* `]`
Executes an extension and gives it control of the `kdbx` session until it quits. The variable *extension* specifies the named extension file and passes arguments to it as specified by the variable *arguments*. Valid options are as follows:

`-debug`
Causes I/O to and from the extension to be printed on the screen.

`-pipe` *in_pipe out_pipe*
Used in conjunction with the `dbx` debugger for debugging extensions. See Chapter 4 for information on using the `-pipe` option.

`-print_output`
Causes the output of the extension to to be sent to the invoker of the extension without interpretation as `kdbx` commands.

`-redirect_output`
Used by extensions that execute other extensions to receive the output from the called extensions. Otherwise, the user receives the output.

`-tty`
Causes `kdbx` to communicate with the subprocess through a tty line instead of pipes. If the `-pipe` option is present, `proc` ignores it.

`print` *string*
Prints *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 `-x` option is present, commands are printed out as

they are executed.

`unalias` *name*
Removes the alias, if any, from *name*.

2.2.4 Predefined Aliases

The kdbx debugger contains many predefined aliases, which are defined in the kdbx startup file (`/var/kdbx/system.kdbxrc`). Table 2-2 lists some of the more commonly used aliases and their definitions.

Table 2-2: kdbx Aliases

Alias	Definition
arp	"proc" arp
array_action	"proc" array_action
buf	"proc" buf
buf_action	list_action "struct buf *" b_forw buf buf
callout_action	list_action "struct callout *" c_next 0 callout
cast	"proc" cast
config	"proc" config
convert	"proc" convert
dis	"proc" dis
echo	"proc" echo
export	"proc" export
fields	"proc" fields
file	"proc" file
h	help
inpcb_action	list_action "struct inpcb *" inp_next
list_action	"proc" list_action
mount_action	list_action "struct mount *" m_next rootfs rootfs
mount	"proc" mount
namecache	"proc" namecache
ofile	"proc" ofile
paddr	"proc" paddr
pcb	"proc" pcb
pr	"proc"
printf	"proc" printf
proc	"proc" proc
procaddr	"proc" procaddr
procp	"proc" -pipe /tmp/pipein /tmp/pipeout
procpd	"proc" -debug -pipe /tmp/pipein /tmp/pipeout

Table 2-2: (continued)

Alias	Definition
proc_action	list_action "struct proc *" p_nxt 0 allproc
ps	"dbx" kps
sh	"proc" -print_output -tty
socket	"proc" socket
sum	"proc" sum
swap	"proc" swap
task	"proc" task
thread	"proc" thread
u	"proc" u
ucred	"proc" ucred
unaliasall	"proc" unaliasall
vnode	"proc" vnode

2.2.5 Extensions

Table 2-3 lists the default extensions for the kdbx debugger that reside in the directory /var/kdbx.

Table 2-3: kdbx Extensions

Extension	Action
arp	Prints contents of the address resolution protocol (arp) table
array_action	Performs some action on each element of an array
buf	Prints the buffer table
callout	Prints the callout table
cast	Tells dbx to print a piece of memory as a given type
config	Displays the configuration of the machine
convert	Converts a number from one base to another
dis	Disassembles instructions
export	Displays the exported file systems
file	Displays the file table
inpcb	Displays the user datagram protocol (udb) and the transmission control protocol (tcp) tables
list_action	Performs some action on each element of a list.
mount	Prints the mount table.
namecache	Prints all namecaches.
ofile	Prints the open files of processes.
paddr	Converts a range of memory to symbolic references.
pcb	Displays the pcb of a process.
printf	Uses the dbx printf capability.
proc	Prints the process table.

Table 2-3: (continued)

Extension	Action
procaddr	Converts an address to a procedure name.
socket	Displays the sockets in the file table.
sum	Displays a summary of the system.
swap	Displays a summary of swap space.
task	Displays all task structures on the system.
thread	Displays all thread structures on the system.
u	Displays a u structure.
ucred	Displays or checks references to ucred structures.
unaliasall	Removes all aliases.
vnode	Displays the vnode table.

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
k	kseg	fffffc00	k0x00487c48
u	user space	00000000	u0x86406200
?	Unrecognized or random type		?0x3782cc33

The following list describes each of the kdbx extensions.

arp

arp[-]

The arp extension prints the contents of the address resolution protocol (arp) table. If the optional hyphen (–) is present, arp prints out the entire arp table; otherwise, it prints out those entries that have nonzero at iaddr.s_addr and at_flags fields.

Example:

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

array_action

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

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 print out specific components or values as described in the list of command options. 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 either 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]`.

options
If the `–head` option is specified, the next argument is printed as the table header.

If the `–size` option is specified, the next argument is used as the array element size. Otherwise, the size is calculated from the element type.

If the `–cond` option is specified, 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:

`%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

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)
```

buf

buf[*addresses* | **–free** | **–all**]

The `buf` extension prints out the buffer table. If no arguments are specified, the buffers on the hash list are displayed.

If addresses are specified, the buffers at those addresses are displayed. If the `–free` option is specified, the buffers on the free list are displayed. If the `–all` option is specified, buffers on the hash list are displayed first, followed by buffers on the free list.

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)
```

callout

callout

The `callout` extension prints the callout table.

Example:

```
(kdbx) callout
FUNCTION                        ARGUMENT              TIME
ss_process_timeouts             0xffffffff90864000       0
thread_timeout                  0xffffffff865b6c28      570
realitexpire                    0xffffffff903e21d8      807
realitexpire                    0xffffffff903e2328      344
realitexpire                    0xffffffff903e2b08      516
realitexpire                    0xffffffff903e3828      889
ubc_dirty_memory                0x0000000000000000      320
   •
   •
   •
```

cast

cast *address type*

The `cast` extension forces `dbx` to print a piece of memory as a given type. This is equivalent to the following command:

`dbx print *((type) address)`

Example:

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

config

config

The `config` extension prints out the configuration of the machine.

Example:

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

(kdbx)
```

convert

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

The `convert` extension converts numbers from one base to another. The `–in` and `–out` options specify the input and output bases, respectively. If `–in` is not present, the input base is inferred from the arguments. The arguments can be either numbers or variables.

Example:

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

dis

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

The `dis` extension disassembles some number of instructions as specified in *`num-instructions`*, starting at *`start-address`*. If the number of instructions is not given, 1 is assumed.

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)
```

export

export

The export extension prints out the exported entries that are mounted remotely.

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/satish
0xffffffff863bfe00  8  2050  15263  731712009   -2     0 /usr/staff/jjchen
0xffffffff863bfe80  8  1024   6528  731270099   -2     0 /mnt
```

file

file [*address*]

The `file` extension prints out the file table. If no arguments are present, all file entries with nonzero reference counts are printed. Otherwise, the file entries located at the specified addresses are printed.

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
    •
    •
    •
```

inpcb

inpcb [**–udp**] [**–tcp**] [*address*]

The `inpcb` extension prints the udb and tcb tables. If no arguments are specified, both tables are printed. If either `–udp` or `–tcp` are present, the corresponding table is printed.

If addresses are present, then `–udp` and `–tcp` are ignored and the entries located at the specified addresses are printed.

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
osfdec.dec.com    6000 decosf.dec.com    1451 v0x8643f408 v0x863da408
osfdec.dec.com     998 decosf.dec.com    1020 v0x8643fc08 v0x863da208
osfdec.dec.com     999 decosf.dec.com     514 v0x8643ac08 v0x8643d008
osfdec.dec.com    6000 decosf.dec.com    1450 v0x863fba08 v0x863dad08
osfdec.dec.com    1008 decosf.dec.com    1021 v0x86431e08 v0x86414708
osfdec.dec.com    1009 decosf.dec.com     514 v0x86412808 v0x8643ce08
osfdec.dec.com    6000 decosf.dec.com    1449 v0x86436608 v0x86415e08
osfdec.dec.com    6000 decosf.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
```

list_action

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

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 print particular components. The argument 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 *end-addr* is 0. If the list is circular, the value of *end-addr* is equal to *start-addr*.

start_addr
The address of the list. This can be either a variable name or a number.

options
If the –head *header* option is specified, the *header* argument is printed as the table header.

If the –cond *arg* option is specified, the *arg* 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 list.

Note

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

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

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

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
   •
   •
   •
```

mount

mount [**–s**] [*address*]

The mount extension prints the mount table. The –s option outputs a short form of the table. If addresses are present, the mount entries named by the addresses are printed.

Example:

```
(kdbx) mount
 MOUNT        MAJ  MIN    VNODE        ROOTVP      TYPE       PATH          \
                   FLAGS
v0x863abab8    8  1024   u0x00000000  v0x90342ba8  ufs    /                 \
                  loc
v0x863aa000            v0x903033a0  v0x903188d8  nfs    /share/firstout/bui\
ld/agosminor.dsk4
v0x863aa1c8            v0x90277798  v0x90315748  nfs    /share/bigbld/build\
/agos.dsk1
v0x863aa390            v0x902771e0  v0x902773c8  nfs    /share/buffer/build\
/submits.dsk2
v0x863aa558            v0x90276e10  v0x90330fa0  nfs    /share/bigbld/build\
/alpha.dsk5
v0x863aa720            v0x90276858  v0x9031ff08  nfs    /share/lastin/build\
/goldos.dsk6
v0x863aa8e8            v0x902762a0  v0x9033e8f0  nfs    /share/bigbld/build\
/agos.dsk2
v0x863aaab0            v0x90275b00  v0x903412e0  nfs    /share/buffer/build\
/submits.dsk1
v0x863aac78            v0x90273e68  v0x903385d0  nfs    /share/lastin/build\
/agosminor.dsk6
v0x863aae40            v0x90269e48  v0x9026cc08  nfs    /usr/comet
v0x863ab008            v0x90266700  v0x9033fde8  nfs    /usr/sde
v0x863ab1d0            v0x90247d18  v0x90269a78  nfs    /usr/staff/alpha1/j\
jchen
v0x863ab398            v0x902692d8  v0x902694c0  nfs    /usr/projects/osf_v\
1.2
v0x863ab560    8     2  v0x90221b18  v0x90322cc8  ufs    /usr3             \
                  loc
v0x863ab728    8  2050  v0x9021f310  v0x903414c8  ufs    /usr              \
                  loc
v0x863ab8f0    8  1025  v0x902209f0  v0x90341c68  ufs    /usr2             \
                  loc
(kdbx)
```

namecache

namecache

The `namecache` extension prints the namecache structures on the system.

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
   •
   •
   •
```

ofile

ofile [**–proc** *address* | **–pid** *pid* | **–v**]

The `ofile` extension prints the open files of processes. If no arguments are specified, the extension prints the files opened by each process. If either `–proc` *`address`* or `–pid` *`pid`* is present, the open files of the given process are printed. The `–v` option prints out more information about the open files.

Example:

```
(kdbx) ofile -pid 1136 -v

Proc=0xffffffff9041e980   pid= 1136

 ADDR_FILE   f_cnt ADDR_VNODE   V_TYPE V_TAG   USECNT HLDCNT   V_MOUNT      INO#   QSIZE
=========== ===== =========== ====== ====== ====== ====== =========== ====== =====
v0x90408520    27  v0x902c1390  VCHR  VT_UFS     3      0   v0x863abab8   1103      0
v0x90408520    27  v0x902c1390  VCHR  VT_UFS     3      0   v0x863abab8   1103      0
v0x90408520    27  v0x902c1390  VCHR  VT_UFS     3      0   v0x863abab8   1103      0
v0x90408368     1  v0x9026e6b8  VDIR  VT_UFS    18      1   v0x863ab728  64253    512
```

paddr

paddr *address number-of-longwords*

The `paddr` extension converts a range of memory to symbolic references. The arguments to the `paddr` extension are as follows:

`address`
The starting address.

`number-of-longwords`
The number of longwords to dump out.

Example:

```
(kdbx) paddr 0xffffffff90be36d8 20
[., 0xffffffff90be36d8]: [h_kmem_free_memory_:824, 0xfffffc000037f47c] 0x0\
000000000000000
[., 0xffffffff90be36e8]: [., 0xffffffff8b300d30] [hardclock:394, 0xfffffc0\
0002a7d5c]
[., 0xffffffff90be36f8]: 0x0000000000000000 [., 0xffffffff863828a0]
[., 0xffffffff90be3708]: [setconf:133, 0xfffffc00004949b0] [., 0xffffffff9\
0be39f4]
[., 0xffffffff90be3718]: 0x00000000000004e0 [thread_wakeup_prim:858, 0xfff\
ffc0000328454]
[., 0xffffffff90be3728]: 0x0000000000000001 0xffffffff0000000c
[., 0xffffffff90be3738]: [., 0xffffffff9024e518] [hardclock:394, 0xfffffc0\
0002a7d5c]
[., 0xffffffff90be3748]: 0x00000000004d5ff8 0xfffffffffffffd4
[., 0xffffffff90be3758]: 0x00000000000bc688 [setconf:133, 0xfffffc00004946\
f0]
[., 0xffffffff90be3768]: [thread_wakeup_prim:901, 0xfffffc00003284d0] 0x00\
0003ff85ef4ca0
```

pcb

pcb *thread_address*

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

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
```

printf

printf *format-string* [*args*]

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

format-string
A character string combining literal characters with conversion specifications.

args
The arguments whose values are to be printed.

Example:

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

proc

proc [*address*]

The `proc` extension prints the process table. If addresses are present, the proc structures at only those locations are printed. Otherwise, all proc structures are printed.

Example:

```
(kdbx) proc
Addr        PID   PPID  PGRP  UID   PY CPU SIGS     Event       Flags
v0x9041f6a0  1166  1136  1136    0   0   0 00000000 v0x863d6e68 in pagv ctty
v0x9041f550  1164  1136  1136    0   0   0 00000000 v0x863d7668 in pagv ctty
v0x9041f400  1163  1136  1136    0   0   0 00000000 v0x863d7e68 in pagv ctty
v0x9041f2b0  1161  1136  1136    0   0   0 00000000 v0x8637a668 in pagv ctty
v0x9041dc60  1098  1097  1097 1138   0   0 00000000 v0x8640f268 in pagv ctty
v0x9041d480  1097  1009  1097 1138   0   0 00000000 v0x8641ec00 in pagv ctty
v0x9041d720  1061  1060  1060 1138   0   0 00000000 v0x8641f668 in pagv ctty
v0x9041cf40  1060  1026  1060 1138   0   0 00000000 v0x8641fe00 in pagv ctty
v0x9041e2f0  1050  1010  1050    0   0   0 00000000 k0x0045f860 in omask pagv ctty
v0x9041d870  1032  1011  1032 1138   0   0 00000000 k0x0045f860 in omask pagv ctty
v0x9041d5d0  1026  1024  1026 1138   0   0 00000000 k0x0045f860 in omask pagv ctty
   .
   .
   .
```

procaddr

procaddr [*address*]

The `procaddr` extension converts an *address* to a procedure name.

Example:

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

socket

socket

The `socket` extension prints out those files from the file table that are sockets with nonzero reference counts.

Example:

```
(kdbx) socket
  Fileaddr   Sockaddr   Type        PCB     Qlen Qlim Scc  Sproc       Rcc  Rproc
========== ========== ===== ========== ==== ==== === ========== ==== ========
v0x904061b8 v0x863b5c08 DGRAM v0x8632dc88   0    0   0  v0x863b5cf8 0 v0x863eff08
v0x90406370 v0x863b5a08 DGRAM v0x8632db08   0    0   0  v0x863b5af8 0 v0x863efef0
v0x90406478 v0x863b5808 DGRAM v0x8632da88   0    0   0  v0x863b58f8 0 v0x863b5898
v0x904064d0 v0x863b5608 DGRAM v0x8632d688   0    0   0  v0x863b56f8 0 v0x863efc68
v0x904065d8 v0x863b5408 DGRAM v0x8632dc08   0    0   0  v0x863b54f8 0 v0x863efc50
v0x90406630 v0x863b5208 DGRAM v0x8632d588   0    0   0  v0x863b52f8 0 v0x863eff80
v0x904067e8 v0x863b4208 DGRAM v0x8632d608   0    0   0  v0x863b42f8 0 v0x863effc8
v0x90406840 v0x863b4008 DGRAM v0x8632d788   0    0   0  v0x863b40f8 0 v0x863b4098
v0x904069a0 v0x8641f008  STRM v0x8632c808   0    0   0  v0x8641f0f8 0 v0x8641f098
v0x90406aa8 v0x863b4c08  STRM v0x8632d508   0    2   0  v0x863b4cf8 0 v0x863efdd0
v0x90406bb0 v0x863b4e08  STRM v0x8632da08   0    0   0  v0x863b4ef8 0 v0x863b4e98
   .
   .
   .
```

sum

sum

The `sum` extension prints a summary of the system.

Example:

```
(kdbx) sum
Hostname : decosf.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)
```

swap

swap

The `swap` extension prints a summary of swap space.

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)
```

task

task [*proc_address*]

The `task` extension prints the task table. If addresses are present, the task structures named by the argument addresses are printed. Otherwise, all tasks are printed.

Example:

```
(kdbx) task
Task Addr    Proc Addr    Ref  Threads  Utask Addr
==========  ========== =====  =======  ==========
v0x8637eec0  v0x9041ead0     3        1  v0x8645a880
v0x8637e440  v0x9041e830     3        1  v0x8645afc0
v0x8637e1a0  v0x9041eec0     3        1  v0x8645b700
v0x86380ba0  v0x9041db10     3        1  v0x86417a00
v0x86380e40  v0x9041d9c0     3        1  v0x86418140
v0x8637ec20  v0x9041e6e0     3        1  v0x863cc140
v0x8637f400  v0x9041ed70     3        1  v0x863cc880
v0x8637f160  v0x9041e980     3        1  v0x863ccfc0
v0x863818c0  v0x9041dc60     3        1  v0x863e8000
   •
   •
   •
```

thread

thread [*proc_address*]

The `thread` extension prints information about threads. If addresses are present, the thread structures named by the addresses are printed. Otherwise, all threads are printed.

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
   •
   •
   •
```

u

u [*proc-addr*]

The u extension prints a u structure. If no argument (*proc-addr*) is specified, the u structure of the currently running process is printed.

Example:

```
(kdbx) u ffffffff9027ff38
procp   0x9027ff38
ar0     0x90c85ef8
comm     cfgmgr
args      gB*  |
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)
```

ucred

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

The `ucred` extension prints all instances of references to ucred structures. The options are described as follows.

Argument	Description
(none)	Prints all ucred references
-proc	Prints all ucreds referenced by the proc structures
-uthread	Prints all ucreds referenced by the uthread structures
-file	Prints all ucreds referenced by the file structures
-buf	Prints all ucreds referenced by the buf structures
-ref address	Prints all references to a given ucred
-check address	Checks the reference count of a particular ucred
-checkall	Checks the reference count of all ucreds (mismatch marked by *)

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
```

unaliasall

unaliasall

The `unaliasall` extension removes all aliases, including the predefined aliases described in Section 2.2.4.

Example:

```
(kdbx) unaliasall
```

vnode

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

The `vnode` extension prints the vnode table. The arguments are described as follows.

Argument	Description
(none)	Prints ACTIVE entries in the vnode table. (ACTIVE means that usecount is nonzero or holdcnt is nonzero.)
-free	Prints INACTIVE entries in the vnode table.
-all	Prints ALL (both ACTIVE and INACTIVE) entries in the vnode table.
-ufs	Prints all UFS vnodes.
-nfs	Prints all NFS vnodes.
-cdfs	Prints all CDFS vnodes.
-fs address	Prints vnode entries of a mounted file system.
-u uid	Prints vnode entries of a particular user.
-g gid	Prints vnode entries of a particular group.
-v	Prints related inode/rnode/cdnode info (used with -ufs, -nfs or -cdfs only).

Example:

```
(kdbx) vnode
ADDR_VNODE  V_TYPE  V_TAG USECNT HLDCNT  V_MOUNT
========== ======= ====== ====== ====== ==========
v0x9021e000  VBLK  VT_NON      1      0  k0x00467ee8
v0x9021e1e8  VBLK  VT_NON     83     18  v0x863abab8
v0x9021e3d0  VBLK  VT_NON      1      0  k0x00467ee8
v0x9021e5b8  VDIR  VT_UFS     34      1  v0x863abab8
v0x9021e7a0  VDIR  VT_UFS      1      1  v0x863abab8
v0x9021ed58  VBLK  VT_UFS      1      0  v0x863abab8
v0x9021ef40  VBLK  VT_NON      1      0  k0x00467ee8
v0x9021f128  VREG  VT_UFS      3      0  v0x863abab8
v0x9021f310  VDIR  VT_UFS      1      1  v0x863abab8
v0x9021f8c8  VREG  VT_UFS      1      0  v0x863abab8
v0x9021fe80  VREG  VT_UFS      1      0  v0x863abab8
v0x902209f0  VDIR  VT_UFS      1      1  v0x863abab8
v0x90220fa8  VBLK  VT_UFS      9      8  v0x863abab8
v0x90221190  VBLK  VT_NON      1      0  k0x00467ee8
v0x90221560  VREG  VT_UFS      1      0  v0x863abab8
v0x90221748  VBLK  VT_UFS   3153    257  v0x863abab8
   •
   •
   •
(kdbx) vnode -nfs -v
ADDR_VNODE  V_TYPE  V_TAG USECNT HLDCNT  V_MOUNT     FILEID  MODE  UID  GID  QSIZE
========== ======= ====== ====== ====== ========== ======= ====== ==== ==== ======
v0x90246820  VDIR  VT_NFS      1      0  v0x863ab560 205732  40751  1138   23   2048
v0x902471a8  VDIR  VT_NFS      1      0  v0x863ab398 378880  40755  1138   10   5120
v0x90247578  VDIR  VT_NFS      1      0  v0x863ab1d0      2  40755     0    0   1024
v0x90247948  VDIR  VT_NFS      1      0  v0x863ab008 116736  40755  1114    0    512
v0x9026d1c0  VDIR  VT_NFS      1      0  v0x863ab1d0  14347  40755     0   10    512
v0x9026e8a0  VDIR  VT_NFS      1      0  v0x863aae40      2  40755     0   10    512
v0x9026ea88  VDIR  VT_NFS      1      0  v0x863ab1d0  36874  40755     0   10    512
v0x90272788  VDIR  VT_NFS      1      0  v0x863ab1d0  67594  40755     0   10    512
v0x902fd080  VREG  VT_NFS      1      0  v0x863ab1d0  49368 100755  8887  177 455168
v0x902ff888  VREG  VT_NFS      1      0  v0x863ab1d0  49289 100755  8887  177 538200
v0x90326410  VREG  VT_NFS      1      0  v0x863aae40 294959 100755     3    4 196608
   •
   •
   •
(kdbx) vnode -ufs -v
ADDR_VNODE  V_TYPE  V_TAG USECNT HLDCNT  V_MOUNT     INODE#  MODE  UID  GID  QSIZE
========== ======= ====== ====== ====== ========== ======= ====== ==== ==== ======
v0x9021e5b8  VDIR  VT_UFS     34      1  v0x863abab8      2  40755     0    0   1024
v0x9021e7a0  VDIR  VT_UFS      1      1  v0x863abab8   1088  40755     0    0   2560
v0x9021ed58  VBLK  VT_UFS      1      0  v0x863abab8   1175  60600     0    0      0
v0x9021f128  VREG  VT_UFS      3      0  v0x863abab8   7637 100755     3    4 147456
v0x9021f310  VDIR  VT_UFS      1      1  v0x863abab8   8704  40755     3    4    512
v0x9021f8c8  VREG  VT_UFS      1      0  v0x863abab8   7638 100755     3    4  90112
v0x9021fe80  VREG  VT_UFS      1      0  v0x863abab8   7617 100755     3    4 196608
v0x902209f0  VDIR  VT_UFS      1      1  v0x863abab8   9792  41777     0   10    512
v0x90220fa8  VBLK  VT_UFS      9      8  v0x863abab8   1165  60600     0    0      0
v0x90221560  VREG  VT_UFS      1      0  v0x863abab8   7635 100755     3    4 245760
v0x90221748  VBLK  VT_UFS   3151    257  v0x863abab8   1184  60600     0    0      0
   •
   •
   •
```

2.3 The kdebug Debugger

The `kdebug` debugger is used for the symbolic, breakpoint debugging of the kernel. Kernels are tested on a DEC OSF/1 test system. The user interface, the `dbx` debugger, runs remotely on a second DEC OSF/1 system, the build system. The build system minimally needs to have a copy of the kernel you are testing and, preferably, the source used to build the kernel. The `dbx` debugger communicates with the test system by connecting the two systems with a serial line. However, a gateway system can be used if it is not possible to locate the test and build systems within a serial line's reach. The build system will then communicate with the gateway system over the internet, and the gateway system will communicate with the test system over the serial line.

2.3.1 Requirements

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

- Test system

  Must be running DEC OSF/1 Version 2.0 or higher, must have the "Kernel Debugging Tools" subset loaded, and must have the "Kernel Breakpoint Debugger" kernel option configured.

- Gateway system

  Must be running DEC OSF/1 Version 2.0 or higher and must have the "Kernel Debugging Tools" subset loaded.

- Build system

  Must be running DEC OSF/1 Version 2.0 or higher and must have the "Kernel Debugging Tools" subset loaded.

2.3.2 Setup

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

- Attach one end of the serial line to the test system.

  Attach the other end of the serial line to the equivalent port on the build system (or gateway system, if one is being used).

  Serial lines are attached to the DEC 3000-series and DEC 4000-series systems using a 25-pin connector slot. On DEC 3000-series systems, this slot is marked by the communications/printer icon.

- To debug kernels, your kernel must be configured with the configuration file option "OPTIONS KDEBUG". If you are debugging the installed kernel, this can be done by selecting "KERNEL BREAKPOINT

DEBUGGING'' from the kernel options menu.

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

2.3.3 Invoking the kdebug Debugger and Using Its Commands

Prior to invoking the `kdebug` debugger, you must install the Product Authorization Key (PAK) for the Developer's kit (OSF-DEV).

You invoke the `kdebug` debugger as follows:

1. On the build system, add the following lines to your `.dbxinit` file if you need to override the default values:

   ```
   set $kdebug_host="gateway_system"
   set $kdebug_line="serial_line"
   set $kdebug_dbgtty="tty"
   ```

   The variable `$kdebug_host` is the name of the gateway system. By default, `$kdebug_host` is set to `localhost`, assuming no gateway system is being used. The variable `$kdebug_line` specifies the serial line to use as defined in the `/etc/remote` file of the build system (or the gateway system, if one is being used). By default, `$kdebug_line` is set to `kdebug`. The variable `$kdebug_dbgtty` sets the tty on the gateway system to display the communication between the build and test systems, which is useful in debugging your setup. To determine the tty name to supply to the `$kdebug_dbgtty` varible, issue the `tty` command in the desired window on the gateway system. By default, `$kdebug_dbgtty` is null.

2. Copy the kernel to be tested to `/vmunix` on the test system.

3. Start 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 may result in unpredictable behavior.

4. Halt the test system and, at the console prompt, set the `boot_osflags` console variable to contain the `k` option, and then boot the system. For

example:

```
>>> set boot_osflags k
>>> b
```

The kernel starts executing, `dbx` will catch the breakpoint, 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 boot your test kernel, see Section 2.3.4 for information on debugging your setup.

Note

By default, the `dbx` debugger assumes the disk copy of the kernel contains accurate instructions. However, if you are debugging portions of the kernel containing self-modifying code (for example, `spl` routines), this is not the case. To handle these situations, you need to add the following line to your `dbxinit` file:

```
set $readtextfile=0
```

Note that setting this variable may degrade the performance of the debugger.

2.3.4 Debugging Your Setup

If you have completed the `kdebug` setup as described in Section 2.3.3 and it stills fails to work, follow the steps in the following list to isolate the problem:

1. Test the serial line connection. Log onto 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 or login getty, may be using the serial line port that you have dedicated to the `kdebug` debugger, for example, `/dev/tty00`. To check for this condition, do the following:

   - Look at the `/etc/inittab` file to see if any processes are using that line. If so, disable these lines until you are finished with the `kdebug` session. See the `inittab`(4) reference page for information on disabling lines.
   - Use the `ps` command to see if any processes are currently using the

line as follows:

```
# ps agxt00
```

If a process is using `tty00`, kill the process using the process ID in the first field.

- See if any runaway `kdebugd` gateway daemons are still running:

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

  If one is running, kill the process using the process ID in the first field.

2. If you detect no problems in step 1, at the console prompt of the test system, ensure that the `boot_osflags` console environment variable specifies the `k` flag. Boot the test system. If the system boots to single user or beyond, then `kdebug` has not been configured into the kernel as specified in Section 2.3.2.

   If the system does not boot and you do not see informational messages in your tip session, then the serial line is not working or is not attached properly. If you see informational messages in the tip session, then the test system and serial line are operating correctly. Exit the tip session.

3. Determine which pseudoterminal 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 pseudoterminal `/dev/ttyp2`. Edit your `$HOME/.dbxinit` file on the build system and make the following edits:

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

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

   - Set the variable `$kdebug_host` to the host name of the system from which you issued the `tip` command. For example, if the host name is DECOSF, the entry in the `.dbxinit` file will be as follows:

     ```
     set $kdebug_host="decosf"
     ```

   - 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 pseudoterminal line `/dev/ttyp2` that `kdebug` is starting. If no messages appear, ensure that the `inetd` daemon is running on the gateway system. Also, check the tcp/ip connection between the build and gateway system using one of the following commands: `rlogin`, `rsh`, or `rcp`.

2.4 The crashdc Utility

The `crashdc` utility collects critical data from operating system crash dump files or from a running kernel. The data collected can be used to analyze the cause of the crash. The `crashdc` utility uses existing system tools and utilities to extract information from crash dumps. The information garnered from crash dumps 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, a data collection file is created with the same numerical file name extension as the crash dump (see Section 1.2.3 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*

The following example shows sample output from `crashdc`:

```
#
# CANASTA Data Collection (Version 1.3)
#
_crash_data_collection_time: Thu Sep  2 15:01:07 EDT 1993
_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 T2.0-1  (Rev. 114.2); Wed Sep  1 09:24:01 EDT 1993
_crash_version: DEC OSF/1 T2.0-1  (Rev. 114.2); Wed Sep  1 09:24:01 EDT 1993
dbx version 3.11.4
Type 'help' for help.

stopped at
warning: Files compiled -g3: parameter values probably wrong
  [boot:1118 ,0xfffffc0000374a08]  Source not available
_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 = "T2.0"
    version = "114.2"
    machine = "alpha"
}
_cpu:  30
_system_string:  0xfffffc0000442fa8 = "DEC3000 - M500"
```

```
_num_cpus:  1
_partial_dump:  1
_physmem(MBytes):  96
_panic_string:  0xfffffc000043cf70 = "kernel memory fault"
_stack_trace_begin:
>  0 boot(reason = 0, howto = 0) ["../../../../src/kernel/arch/alpha/machdep.c":\
1118, 0xfffffc0000374a08]
   1 panic(s = 0xfffffc000043cf70 = "kernel memory fault") ["../../../. ./src/ke\
rnel/bsd/subr_prf.c":616, 0xfffffc000024ff60]
   2 trap() ["../../../../src/kernel/arch/alpha/trap.c":945, 0xfffffc0000381440]
   3 _XentMM() ["../../../../src/kernel/arch/alpha/locore.s":949, 0xfffffc000037\
2dec]
_stack_trace_end:
_preserved_message_buffer_begin:
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:
  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
_current_tid:  0xffffffff863d36c0
_proc_thread_list_begin:
thread 0x863d36c0 stopped at [boot:1118,0xfffffc0000374a08] Source not available
_proc_thread_list_end:
_dump_begin:
>  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/ker\
nel/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, 0xfffffc000037\
2dec]

_dump_end:
_kernel_thread_list_begin:
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]  Sourc\
e not available
thread 0x8632d110 stopped at [thread_block:1400 +0x1c,0xfffffc00002ca2f8]  Sourc\
e not available
_kernel_thread_list_end:
_savedefp:  0xffffffff96199940
_kernel_memory_fault_data_begin:
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

dbx version 3.11.4
Type 'help' for help.

stopped at
warning: Files compiled -g3: parameter values probably wrong
  [boot:1118 ,0xfffffc0000374a08]  Source not available
```

```
_savedefp_exception_frame_(savedefp/33X):
fffffffff96199940:  0000000000000000 fffffc000046f888
fffffffff96199950:  ffffffff863d36c0 0000000079c2c93f
   •
   •
   •
fffffffff96199a30:  0000000000901402 0000000000001001
fffffffff96199a40:  0000000000002000
_savedefp_exception_frame_ptr:  0xfffffffff96199940
_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 Output (swap,sum)
#
dbx version 3.11.4
Type 'help' for help.

stopped atwarning: Files compiled -g3: parameter values probably wrong
  [boot:1118 ,0xfffffc0000374a08]  Source not available
_kdbx_sum:
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_swap:
       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_proc:
Addr         PID  PPID  PGRP  UID PY CPU SIGS     Event       Flags
v0x95aaf6a0   489  340   489    0  0   0 00000000 NULL in pagv ctty
v0x95aad5d0   342  337   342    0  0   0 00000000 v0x8632fdf0 in pagv ctty
v0x95aad8f0   341  337   341    0  0   0 00000000 v0x8632cdc8 in pagv ctty
   •
   •
   •
v0x95aad2b0     1    0     1    0  0   0 00000000 k0x0048dc10 in omask pagv
v0x95aad120     0    0     0    0  0   0 00000000 v0x8632fdf0 in sys
#
_crash_data_collection_finished:
```

Crash Analysis Examples 3

This chapter provides examples of how to examine crash dumps using the tools discussed in Chapter 2.

3.1 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, the following guidelines should assist you in identifying the events that led to the crash:

- Gather some facts about the system (for example, operating system type, version number, revision level, hardware configuration).
- Look at the panic string, if one exists. This string is contained in the preserved message buffer (`pmsgbuf`) and in the `panicstr` global variable.
- Locate the thread executing at the time of the crash. Most likely, this thread will contain the events that lead to the panic.
- Determine whether you can fix the problem. If the system crashed because of lack of resources (for example, swap space), you can probably eliminate the problem by adding more of that resource.

  If the problem is with the software, you may need to file a Software Performance Report (SPR).

3.2 Identifying a Software Panic with the dbx Debugger

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 Print 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.

3.3 Identifying a Hardware Trap with the dbx Debugger

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 = "decosf.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
00020   update
01604   getty
00099   syslogd
00101   binlogd
00195   nfsd
00155   ypserv
00151   portmap
00194   nfsd
00193   nfsd
00191   mountd
00196   nfsd
00197   nfsd
00198   nfsd
00199   nfsd
00200   nfsd
00201   nfsd
00202   nfsd
00204   nfsiod
00205   nfsiod
00206   nfsiod
00207   nfsiod
00209   rpc.pcnfsd
00211   rpc.statd
00213   rpc.lockd
00219   automount
00226   automount
00230   automount
00234   automount
00241   sendmail
00260   inetd
00265   cron
00293   xdm
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>/: file system full
<3>/: file system 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
<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
<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
NFS write error 70 on host flume fh 182a 9e04 35 2 0 0 0 0
NFS write error 70 on host flume fh 182a 9e04 35 2 0 0 0 0
NFS write error 70 on host flume fh 182a 9e04 35 2 0 0 0 0
<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 dbx debugger's sh command to issue commands to the shell. In this case, issue the stings and grep commands to pull the operating system revision number in the vmunix.1 dump file.

2 Print the utsname structure to obtain some more information about the operating system version.

3 Print 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 panic. However, the stack trace does not show what caused the trap.

5 Look to see what processes were running when the system crashed using 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 Print the preserved message buffer (pmsgbuf). Note the bolded value for the program counter (pc).

8 Print register 28 of the exception frame pointer (savedefp). This register always contains the pc value. The pc value can always be obtained in either pmsgbuf or savedefp.

9 Dissassemble 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 function ubc_invalidate, which resides in file /vfs/vfs_ubc.c at line number 1768, called the function nfs_putpage at line number 2344 in the file /kern/sched_prim.c and the system stopped.

3.4 Debugging Kernel Threads with the dbx Debugger

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, 0xfffffc000\
033961c]
   1 mpsleep(chan = 0xffffffff8d4ef450 = "", pri = 282, wmesg = 0xfffffc000046f29\
0 = "network", timo = 0, lockp = (nil), flags = 0) ["../../../../src/kernel/bsd/k\
ern_synch.c":267, 0xfffffc00002b772c]
   2 sosleep(so = 0xffffffff8d4ef408, addr = 0xffffffff906cfcf4 = "^P", pri = 282\
,tmo = 0) ["../../../../src/kernel/bsd/uipc_socket2.c":612, 0xfffffc00002d3784]
   3 accept1(p = 0xffffffff8f8bfde8, args = 0xffffffff906cfe50, retval = 0xffffff\
ff906cfe40, compat_43 = 1) ["../../../../src/kernel/bsd/uipc_syscalls.c":300, 0xf\
ffffc00002d4c74]
   4 oaccept(p = 0xffffffff8d431c68, args = 0xffffffff906cfe50, retval = 0xffffff\
ff906cfe40) ["../../../../src/kernel/bsd/uipc_syscalls.c":250, 0xfffffc00002d4b0c]
   5 syscall(ep = 0xffffffff906cfef8, code = 99, sr = 1) ["../../../../src/kernel\
/arch/alpha/syscall_trap.c":499, 0xfffffc00003ec18c]
   6 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":675, 0xfffffc00003\
df96c]
(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 0x8d431038 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d430e30 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d430c28 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d430a20 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d430818 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d430610 stopped at   [thread_run:1486 +0x18,0xfffffc00003398e0]        \
Source not available
thread 0x8d430408 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
More (n if no)?
thread 0x8d430200 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d42f9e0 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
```

```
thread 0x8d42f7d8 stopped at   [thread_block:1289 +0x18,0xfffffc00003394b8]      \
Source not available
thread 0x8d42f5d0 stopped at   [boot:696 ,0xfffffc00003e119c]     Source not avai\
lable
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 avail\
able
(dbx) t                      5
>  0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep\
.c":694, 0xfffffc00003e1198]
  1 panic(s = 0xfffffc000048a098 = "    sp contents at time of fault: 0x%l016x\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, 0xfffffc00003df\
b1c]
  4 pmap_release_page(pa = 18446744071785586688) ["../../../../src/kernel/arch/a\
lpha/pmap.c":640, 0xfffffc00003e3ecc]
  5 put_free_ptepage(page = 5033216) ["../../../../src/kernel/arch/alpha/pmap.c"\
:534, 0xfffffc00003e3ca0]
  6 pmap_destroy(map = 0xffffffff8d5bc428) ["../../../../src/kernel/arch/alpha/p\
map.c":1891, 0xfffffc00003e6140]
  7 vm_map_deallocate(map = 0xffffffff81930ee0) ["../../../../src/kernel/vm/vm_m\
ap.c":482, 0xfffffc00003d03c0]
  8 task_deallocate(task = 0xffffffff8d568d48) ["../../../../src/kernel/kern/tas\
k.c":237, 0xfffffc000033c1dc]
  9 thread_deallocate(thread = 0x4e4360) ["../../../../src/kernel/kern/thread.c"\
:689, 0xfffffc000033d83c]
 10 reaper_thread() ["../../../../src/kernel/kern/thread.c":1952, 0xfffffc000033\
e920]
 11 reaper_thread() ["../../../../src/kernel/kern/thread.c":1901, 0xfffffc000033\
e8ac]
(dbx) q
```

1 Print 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. If the `boot` thread does not exist, you will need to 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 function `pmap_release_page` at line 640 of the file `pmap.c`.

3.5 Identifying a Software Panic with the kdbx Debugger

This section contains two examples that show how to examine software panics with kernel debugging tools.

Example 1:

```
# 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 : decosf.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 `kdbx` debugger's `sum` command to get a summary of the system.

2 Print 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.

You can also pull the same information from the `crashdc` utility output file `crash-data.3`. This file contains a significant amount of information from the crash dump files. The following example shows the contents of the `crash-data` file from the crash dumps in the previous example:

```
# more crash-data.3
#
# CANASTA Data Collection (Version 1.2) for DEC OSF/1 V1.2
#
#     Modified for 64 bit 21064 ALPHA RISC platforms 8/1/92
#
_crash_data_collection_time: Mon Dec 14 12:25:26 EST 1992
_current_directory: /
_crash_kernel: /var/adm/crash/vmunix.1
_crash_core: /var/adm/crash/vmcore.1
_crash_arch: alpha
_crash_os: DEC OSF/1
_host_version: DEC OSF/1 T1.2-2  (Rev. 5); Fri Dec 04 10:07:50 EST 1992
_crash_version: DEC OSF/1 T1.2-2  (Rev. 5); Fri Dec 04 10:07:50 EST 1992
dbx version 10.0.1
Type 'help' for help.

stopped at  [boot:753 ,0xfffffc00003c4b04]     Source not available
_crashtime:  struct {
    tv_sec = 724353436
    tv_usec = 136442
}
_boottime:  struct {
    tv_sec = 724352791
    tv_usec = 105408
}
_config:  struct {
    sysname = "OSF1"
    nodename = "decosf.dec.com"
    release = "1.2"
    version = "1.2"
    machine = "alpha"
}
_cpu:  30
_system_string:  0xfffffc000046e920 = "DEC3000 - M500"
_num_cpus:  1
_physmem(MBytes):  192
_panic_string:  0xfffffc0000453ea0 = "wdir: compact2"    1
stack_trace_begin:                                       2
 0 boot(paniced = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep.\
c":753, 0xfffffc00003c4b04]
  1 panic(s = 0xfffffc00002e0938 = "") ["../../../../src/kernel/bsd/subr_prf.c"\
:1119, 0xfffffc00002bdbb0]
  2 direnter(ip = 0xffffffff00000000, ndp = 0xffffffff9d38db60) ["../../../../s\
rc/kernel/ufs/ufs_lookup.c":986, 0xfffffc00002e2adc]
  3 ufs_mkdir(ndp = 0xffffffff9d38a2f8, vap = 0x100000020) ["../../../../src/ke\
rnel/ufs/ufs_vnops.c":2383, 0xfffffc00002e9cbc]
  4 mkdir(p = 0xffffffff9c43d7c0, args = 0xffffffff9d38de50, retval = 0xffffffff\
f9d38de40) ["../../../../src/kernel/vfs/vfs_syscalls.c":2579, 0xfffffc00002fd930
  5 syscall(ep = 0xffffffff9d38def8, code = 136) ["../../../../src/kernel/arch/\
alpha/syscall_trap.c":532, 0xfffffc00003cfa54]
  6 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":703, 0xfffffc0000\
3c3200]
_stack_trace_end:
_preserved_message_buffer_begin: struct {                3
    msg_magic = 405601
    msg_bufx = 1501
    msg_bufr = 1457
    msg_bufc = "Alpha boot: available memory from 0x7f8000 to 0xc000000
DEC OSF/1 T1.2-2  (Rev. 5); Fri Dec 04 10:07:50 EST 1992
```

```
physical memory = 190.00 megabytes.
available memory = 173.96 megabytes.
using 729 buffers containing 5.69 megabytes of memory
tc0 at nexus
scc0 at tc0 slot 7
asc0 at tc0 slot 6
rz0 at asc0 bus 0 target 0 lun 0 (DEC     RZ25     (C) DEC 0700)
rz1 at asc0 bus 0 target 1 lun 0 (DEC     RZ25     (C) DEC 0700)
rz2 at asc0 bus 0 target 2 lun 0 (DEC     RZ25     (C) DEC 0700)
rz3 at asc0 bus 0 target 3 lun 0 (DEC     RZ25     (C) DEC 0700)
rz4 at asc0 bus 0 target 4 lun 0 (DEC     RRD42   (C) DEC  4.5d)
asc1 at tc0 slot 6
rz8 at asc1 bus 1 target 0 lun 0 (DEC     RZ57     (C) DEC 5000)
rz9 at asc1 bus 1 target 1 lun 0 (DEC     RZ57     (C) DEC 5000)
rz10 at asc1 bus 1 target 2 lun 0 (DEC     RZ57     (C) DEC 5000)
rz11 at asc1 bus 1 target 3 lun 0 (DEC     RZ57     (C) DEC 5000)
tz12 at asc1 bus 1 target 4 lun 0 (DEC     TLZ04 1989(C)DEC 1615)
fb0 at tc0 slot 8
1280X1024
ln0: DEC LANCE Module Name: PMAD-BA
ln0 at tc0 slot 7
ln0: DEC LANCE Ethernet Interface, hardware address: 08:00:2b:2b:f6:3b
asc2 at tc0 slot 0
asc3 at tc0 slot 0
fza0 at tc0 slot 2
fza0: DEC DEFZA FDDI Interface, hardware address 08:00:2b:2c:20:ff ROM rev 1.0
Firmware rev 1.2
DEC3000 - M500 system
Firmware revision: 1.1
PALcode: OSF version 1.14
lvm0: configured.
lvm1: configured.
setconf: bootdevice_parser translated 'SCSI 0 6 0 0 0 0 FLAMG-IO' to 'rz0'
panic: wdir: compact2
syncing disks... done
"
}
_preserved_message_buffer_end:
_kernel_process_status_begin:                    4
  PID   COMM
00000    kernel idle
00001    init
00002    device server
00003    exception hdlr
00090    sh
00057    cfgmgr
00059    update
00283    sh
00306    automount
00268    nfsd
00225    ypbind
00152    binlogd
00150    syslogd
00181    named
00161    routed
00222    ypserv
00218    portmap
00265    nfsd
00267    nfsd
00242    sendmail
```

```
00243   mail
00263   mountd
00269   nfsd
   •
   •
   •
00277   nfsiod
00280   rpc.statd
00282   rpc.lockd
00290   automount
00299   automount
00304   automount
_kernel_process_status_end:
_current_pid:  306                                  5
_savedefp:  (nil)
_ep:  0x11fffab8
_uptime: .17 hours

#
# Kdbx Output (swap,sum)
#
dbx version 10.0.1
Type 'help' for help.

stopped at  [boot:753 ,0xfffffc00003c4b04]     Source not available
_kdbx_sum:                                          6
Hostname : decosf.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_swap:                                         7

      Swap device name              Size       In Use       Free
--------------------------------  ----------  ----------  ----------
/dev/rz0g                           219648k       6824k     212824k  Dumpdev
                                     27456p        853p      26603p

/dev/rz1c                           416256k          8k     416248k
                                     52032p          1p      52031p
--------------------------------  ----------  ----------  ----------
Total swap partitions:    2         635904k       6832k     629072k
                                     79488p        854p      78634p
#
_crash_data_collection_finished:
```

1 Contents of the panic string global variable (`panicstr`)

2 Stack trace of the current thread block

3 Contents of the preserved message buffer (`pmsgbuf`)

4 Output from the `dbx` debugger's `kps` command

5 The process identifier (PID)

6 Output from the `kdbx` debugger's `sum` command

7 Output from the `kdbx` debugger's `swap` command

Example 2:

```
# kdbx -k vmunix.4 vmcore.4
dbx version 3.11.1
Type 'help' for help.

stopped at  [boot:799 ,0xfffffc0000373488]     Source not available
(kdbx) p panicstr          1
0xfffffc0000401ce8 = "blkfree: freeing free frag"
(kdbx) t                   2
>  0 boot(reason = 0, arghowto = 0) ["../../../../src/kernel/arch/alpha/machdep.c\
":799, 0xfffffc0000373488]
  1 panic(s = 0xfffffc0000401ce8 = "blkfree: freeing free frag") ["../../../../s\
rc/kernel/bsd/subr_prf.c":1132, 0xfffffc000026ddbc]
  2 blkfree(ip = 0xffffffff8d1179c8, bno = 4547232, size = 2048) ["../../../../s\
rc/kernel/ufs/ufs_alloc.c":1331, 0xfffffc000028cc78]
  3 itrunc(oip = 0xffffffff8c728620, length = 0, flags = 22) ["../../../../src/k\
ernel/ufs/ufs_inode.c":1600, 0xfffffc0000291638]
  4 ufs_inactive(vp = 0xffffffff8c728570) ["../../../../src/kernel/ufs/ufs_inode\
.c":911, 0xfffffc0000290694]
  5 vrele(vp = 0xfffffc00002ac504) ["../../../../src/kernel/vfs/vfs_subr.c":954,\
 0xfffffc00002a9c40]
  6 iput(ip = 0xfffffc0000290504) ["../../../../src/kernel/ufs/ufs_inode.c":760,\
 0xfffffc0000290500]
  7 ufs_remove(ndp = 0xffffffff8d1142f8) ["../../../../src/kernel/ufs/ufs_vnops.\
c":1717, 0xfffffc0000298e58]
  8 unlink(p = 0xffffffff8c76fc70, args = 0xffffffff8d117e50, retval = 0xffffffff\
f8d117e40) ["../../../../src/kernel/vfs/vfs_syscalls.c":1630, 0xfffffc00002ac000]
  9 syscall(ep = 0xffffffff8d117ef8, code = 10) ["../../../../src/kernel/arch/al\
pha/syscall_trap.c":576, 0xfffffc000037e838]
 10 _Xsyscall() ["../../../../src/kernel/arch/alpha/locore.s":710, 0xfffffc00003\
71b00]
(kdbx) q
dbx (pid 160) died.  Exiting...
```

1 Print the panic string (`panicstr`). The `blkfree` function called the `panic` function.

2 Perform a stack trace of the current thread block. The stack trace shows that the `blkfree` function at line 1132 in the `ufs_alloc.c` file called the `panic` function.

3.6 Identifying a Hardware Error with kdbx Debugger

The following example shows a method for identifying a hardware error 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 : decosf.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 `kdbx` debugger's `sum` command to get a summary of the system.

2 Print the panic string (`panicstr`).

3 Perform a stack trace. Because the `kn15aa_machcheck` function called the `panic` function, the system crash was probably the result of a hardware error.

Writing Extensions to the kdbx Debugger 4

To assist in debugging a kernel or subsystem, 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 instructions on creating and compiling extensions and includes template extension files and source programs for extensions included on the system.

The DEC OSF/1 source kit must be installed on your system before you can create custom extensions to the `kdbx` debugger. If the source kit is installed, you can create an extension in the `src/usr/bin/kdbx/extensions` directory of the source kit directory structure and add the name to the `Makefile` in that directory to be able to build an extension. See Section 4.4 for more information.

Note

For the remainder of this chapter, the relative pathname `./` refers to the top the source kit directory structure. For example, if you installed the source kit in the `/usr/source_kit` directory, the relative pathname `./src/usr/bin/kdbx/extensions` refers to the directory `/usr/source_kit/src/usr/bin/kdbx/extensions`.

4.1 Considerations and Guidelines

Before writing an extension, you need to consider the following:

- The information that is needed

  You need to identify the information that is relevant to the debugging process. Look at the source code to determine which variables and symbols to use and examine in coding the extension.

- The means for displaying the information

  The information you gather should be displayed in a manner that is easy to read and can be understood by anyone who needs to use the extension.

- The need to provide useful error checking

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

- The availability of `kdbx` library functions

  The `kdbx` debugger comes with a number of standard library functions that you can use in your extension program. These functions are declared in the file `./src/usr/bin/kdbx/krash.h`. Section 4.2 gives the syntax and briefly describes each library routine.

- The availability of templates and examples discussed in this chapter and the extensions in the source tree as a basis for writing extensions

This chapter includes two template files and three examples to use as guidelines for creating your extension program. These examples are annotated to highlight certain features of the programs.

4.2 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 file `./src/usr/bin/kdbx/krash.h`) to develop customized extensions for your application or subsystem. The following section provides a list of these functions, their syntax, and a brief description.

4.2.1 Standard kdbx Library Functions

The following list describes the standard `kdbx` library functions:

addr_to_proc

char *addr_to_proc(long *addr*);

Argument	Input/Output
addr	Input

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

Usage:

```
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);
```

array_element

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

Argument	Input/Output
sym	Input
i	Input
error	Output

The `array_element()` function returns a representation of the *i*th element of the array *sym*. Returns non-NULL if it succeeded or NULL if an error occurred. When the value of error is non-NULL, the *error* argument is set to point to the error message.

As shown in Example 4-4, the `array_element()` function is usually used 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.

The first argument of the `array_element()` function usually comes from the returned result of 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 (see also `read_sym()`).

Usage:

```
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);
}
```

array_element_val

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

Argument	Input/Output
sym	Input
i	Input
ele_ret	Output
error	Output

The `array_element_val()` function returns, in `ele_ret`, the value of element `i` in array `sym`. Returns TRUE if successful, FALSE otherwise. When the value of error is non-NULL, the *`error`* argument is set to point to the error message.

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.

Usage:

```
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);
}
```

array_size

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

Argument	Input/Output
sym	Input
error	Output

The `array_size()` function returns the size of the array described by `sym`. When the value of error is non-NULL, the *`error`* argument is set to point to the error message.

Usage:

```
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);
}
```

cast

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

Argument	Input/Output
`addr`	Input
`type`	Input
`ret_type`	Output
`error`	Output

The `cast()` function returns, in *`ret_type`*, a `DataStruct` that represents an object whose type is *`type`* and whose address is *`addr`*. If successful, TRUE is returned. Otherwise, *`error`* is filled in and FALSE is returned. When the value of error is non-NULL, the *`error`* argument is set to point to the error message.

The `cast()` function is usually used with the `read_field_vals()` function. Given the value of a pointer to a structure, first you use 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 pointed to by the pointer.

Usage:

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

check_args

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

Argument	Input/Output
argc	Input
argv	Input
help_string	Input

The `check_args()` function displays the contents of `help_string` if the –`help` option is specified in the command line. The `check_args` function should be the first action performed by an extension.

Usage:

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

check_fields

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

Argument	Input/Output
symbol	Input
fields	Input
nfields	Input
hints	Input

The `check_fields()` function verifies that the structure described by `symbol` contains the fields described in `fields` and that they are of the correct type. The `nfields` argument is the size of the `fields` array. If

successful, TRUE is returned. Otherwise, the error parts of the affected fields are filled in with errors, and FALSE is returned. The `hints` argument is unused and should always be set to NULL.

Usage:

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

You should check the structure type using the `check_fields()` function before using the `read_field_vals()` function to read field values of this structure type. The `check_fields()` function needs to be used only once. Even though you may use the `read_field_vals()` function repeatedly inside a loop, the `check_fields()` function needs to be invoked only once before the loop.

Therefore, a normal practice is to invoke, in the beginning of the extension, the `check_fields()` function to verify the structure type that the extension is going to read values from using the `read_field_vals()` function.

context

void context(Boolean *user*);

Argument	Input/Output
user	Input

The `context()` function directs `kdbx` to set `user` context or `proc` context, depending on whether `user` is set to TRUE or FALSE. If `user` is TRUE, aliases defined in the extension affect user aliases.

Usage:

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

dbx

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

Argument	Input/Output
command	Input
expect_output	Input

The `dbx( )` function directs `kdbx` to pass the command to `dbx`. If *`expect_output`* is TRUE, the procedure returns after the command is sent, and expects the extension to read the output from `dbx`. If it is FALSE, the procedure expects no output, reads the acknowledgement from `kdbx`, and returns after the command ends.

Usage:

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

deref_pointer

DataStruct deref_pointer(DataStruct *data*);

Argument	Input/Output
data	Input

The `deref_pointer( )` function returns a representation of the object to which *`data`* points. It prints an error message if *`data`* is not a pointer type.

Usage:

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

field_errors

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

Argument	Input/Output
fields	Input
nfields	Input

The `field_errors()` function prints out error messages in *`fields`*.

Usage:

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

format_addr

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

Argument	Input/Output
addr	Input
buffer	Output

The `format_addr()` function puts a representation of *`addr`* into *`buffer`*. The *`buffer`* argument must be at least 12 characters long.

The `format_addr()` function converts a 64-bit address of type long into a 32-bit address of type string. It is used to save space on the output line. For example, the 64-bit address `0xffffffff12345678` is converted into `v0x12345678`.

Usage:

```
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);
}
```

free_sym

void free_sym(DataStruct *sym***);**

Argument	Input/Output
sym	Input

The `free_sym()` function frees the storage associated with *`sym`*.

Usage:

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

krash

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

Argument	Input/Output
command	Input
quote	Input
expect_output	Input

The `krash()` function causes `kdbx` to evaluate *`command`*. Setting *`quote`* to TRUE causes the characters quote (`"`), apostrophe (`'`), and backslash (`\`) to be appropriately quoted. If *`expect_output`* is TRUE, then the

procedure returns after the command is sent and expects the extension to read the output from kdbx. If it is FALSE, the procedure expects no output, reads the acknowledgement from kdbx, and returns after the command ends.

Usage:

```
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);
}
```

list_nth_cell

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

Argument	Input/Output
addr	Input
type	Input
n	Input
next_field	Input
do_check	Input
val_ret	Output
error	Output

The `list_nth_cell()` function returns, in *`val_ret`*, the address of cell *n* in the list described by the start address *`addr`*, of the cell type *`type`*, and the name of next field *`next_field`*. If the routine succeeds, it returns TRUE. If the routine fails, it returns FALSE and an error message is returned in *`error`*. If the argument *`do_check`* is set to TRUE, the routine validates each of the arguments to ensure that correct information is being supplied. If the argument is FALSE, no checking is done.

Usage:

```
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);
}
```

new_proc

void new_proc(char *_args_, char **_output_ret_);

Argument	Input/Output
args	Input
output_ret	Output

The `new_proc()` function directs `kdbx` to execute a `proc` command with arguments specified in *`args`*. If *`output_ret`* is non-NULL, the output from the command is returned in *`*output_ret`*. Otherwise, the output is lost.

Usage:

```
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);
}
```

next_number

Boolean next_number(char *_buf_, char **_next_, long *_ret_);

Argument	Input/Output
buf	Input
next	Output
ret	Output

The `next_number()` function converts the next token in *`buf`* to an integer and returns it in *`ret`*. If *`next`* is non-NULL, it is set to point in *`buf`* after the number. Returns TRUE if successful, FALSE if there was an error.

Usage:

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

next_token

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

Argument	Input/Output
ptr	Input
len_ret	Output
next_ret	Output

The `next_token()` function returns a pointer to the beginning of the next token in *`ptr`*. A token is a sequence of nonspace characters. If *`len_ret`* is non-NULL, the token length is returned in it. If *`next_ret`* is non-NULL, a pointer to the first character after, but not included in the token, is returned in it.

Usage:

```
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);
}
```

print

void print(char *_message_);

Argument	Input/Output
message	Input

The `print()` function directs `kdbx` to print *`message`*.

Because of the I/O 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 output information on the screen. Although the `fprintf(stderr,...)` 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 output the contents of the buffer to the screen.

Note that the `print()` library function automatically prints a newline at the end of the output and that it will fail if it detects a newline character at the end of the buffer.

Usage:

```
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");
}
```

print_status

void print_status(char *_message_, Status *_status_);

Argument	Input/Output
message	Input
status	Input

The `print_status()` function prints a representation of *`status`* with *`message`*.

Usage:

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

quit

void quit(int *i*);

Argument	Input/Output
i	Input

The `quit()` function sends a `quit` command to `kdbx` and exits with status *i*.

Usage:

```
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);
}
```

read_field_vals

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

Argument	Input/Output
data	Input
fields	Input
nfields	Input

The `read_field_vals()` function reads the values of fields of the structure described by *`data`*. The fields are described in *`fields`*. The argument *`nfields`* is the size of the fields array. If this 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.

Usage:

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

read_line

char *read_line(Status * *status*);

Argument	Input/Output
status	Output

The `read_line()` function returns the next line of the output from the last `kdbx` command executed. NULL is returned at the end of the output or if an error occurred. If `status` equals OK when it returns NULL, then the end of the output was reached. Otherwise, an error occurred.

Usage:

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

read_memory

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

Argument	Input/Output
start_addr	Input
n	Input
buf	Output
error	Output

This routine returns the contents of *n* bytes of memory starting at the address `start_addr`. It can be used to look up any type of value. The `read_memory()` function returns True (1) on success and False (0) on failure.

This function is useful in retrieving the values of pointers that point to other pointers.

Usage:

```
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_memory0);
   fprintf(stderr, "%s0, error);
   quit(1);
}
```

read_response

char *read_response(Status * *status*);

Argument	Input/Output
status	Output

The `read_response()` function returns the response from the last `kdbx` command executed. If any errors occurred, NULL is returned and *`status`*, if it is a non-NULL pointer, is filled in.

Usage:

```
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):0, type);
    quit(1);
  }
  free(resp);
}
```

read_sym

DataStruct read_sym(char * *name*);

Argument	Input/Output
name	Input

The `read_sym()` function returns a representation of the symbol *name*. The symbol name is normally a global pointer to a structure or an array of structures inside the kernel. Often the result returned by the `read_sym()` function is then used as the input argument of the `array_element()`, `array_element_val()`, or `read_field_vals()` function.

Usage:

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

read_sym_addr

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

Argument	Input/Output
name	Input
ret_val	Output
error	Output

The `read_sym_addr()` function returns the address of *name* in *ret_val*. When the value of error is non-NULL, the *error* argument is set to point to the error message.

Usage:

```
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);
}
```

read_sym_val

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

Argument	Input/Output
name	Input
type	Input
ret_val	Output
error	Output

The `read_sym_val()` function returns the value of the symbol *name*. The *type* argument is the expected type of the value. The *ret_val* argument is filled in with the value. When the value of error is non-NULL, the *error* argument is set to point to the error message.

Usage:

```
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);
}
```

The `read_sym_val()` function is used 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.

struct_addr

char *struct_addr(DataStruct *data*);

Argument	Input/Output
data	Input

The `struct_addr()` function returns the address of *data*.

Usage:

```
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("");
}
```

to_number

Boolean to_number(char *_str_, long *_val_);

Argument	Input/Output
str	Input
val	Output

The `to_number()` function converts `str` to a number. Returns TRUE if successful, FALSE if conversion was not possible.

Usage:

```
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();
  }
}
```

4.2.2 Standard kdbx Extension Data Types

The routines in Section 4.2.1 use the following data types: `StatusType`, `Status`, `FieldRec`, and `DataStruct`. The uses of these data types are as follows:

- *StatusType* – used to declare the status type, which can take on any one of the following values:
  - No error - (`OK`)
  - Communication error - (`Comm`)
  - Other type of error - (`Local`)

Usage:

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

- *Status* – 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.

  Usage:

```
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`.

- *FieldRec* – used to declare a field of interest in a data structure.

  Usage:

```
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 declarations `caddr_t data` and `char *error` are initially set to NULL. The `read_field_vals` function fills in these values.

- *DataStruct* – used to declare data structures with opaque data types.

  Usage:

```
typedef long DataStruct;
```

4.3 Examples of kdbx Extensions

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

- Extensions that use lists. Example 4-1 provides a C language template and Example 4-2 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 4-3 provides a C language template and Example 4-4 is the source code for the `/var/kdbx/file` extension, which shows how to develop an extension utilizing arrays.
- Extensions that use global symbols. Example 4-5 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 4-1: Template Extension Using Lists

```
static char *help_string =
"<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;
    }
```

Example 4-1: (continued)

```
    <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 issues 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 that will 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 4-2: Extension that uses linked lists: callout.c

```
#include <stdio.h>
#include "krash.h"

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

FieldRec fields[] = {
  { ".c_time", NUMBER, NULL, NULL },
  { ".c_arg", NUMBER, NULL, NULL },
  { ".c_func", NUMBER, NULL, NULL },
  { ".c_next", NUMBER, NULL, NULL },
};

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

main(int argc, char **argv)
{
  DataStruct callout;
  long next;
  char buf[256], *func, *error, arg[13];

  check_args(argc, argv, help_string);
  if(!check_fields("struct callout", fields, NUM_FIELDS, NULL)){
    field_errors(fields, NUM_FIELDS);
```

Example 4-2: (continued)

```
    quit(1);
  }
  if(!read_sym_val("callout", NUMBER, &next, &error)){
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  sprintf(buf, "FUNCTION                          ARGUMENT         TIME");
  print(buf);
  do {
    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, fields, NUM_FIELDS)){
      field_errors(fields, NUM_FIELDS);
      break;
    }
    func = addr_to_proc((long) fields[2].data);
    format_addr((long) fields[1].data, arg);
    sprintf(buf, "%-32.32s %s %10u", func, arg,
            fields[0].data);
    print(buf);
    next = (long) fields[3].data;
  } while(next != 0);
  quit(0);
}
```

Example 4-3: Template Extensions Using Arrays

```
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) ||
```

Example 4-3: (continued)

```
    fprintf(stderr, "Couldn't read size:\n");
    fprintf(stderr, "%s\n", error);
    quit(1);
  }
  <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 element0);
     fprintf(stderr, "%s0, error);
     return(False);
      }
      <print fields in this element> 7
    }
  }
}
```

1 The help string is output by the `check_args` function if the user issues 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 that will 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]")`

- Hardcoding the array size.

6 Specifies the table header string.

7 Extracts, formats, and prints the field information.

Example 4-4: Extension that uses arrays: file.c .

```
#include <stdio.h>
#include <sys/fcntl.h>
#include "krash.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\
";

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 }
};

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

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

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

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,
```

Example 4-4: (continued)

```
          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);
}

static Boolean prfiles(DataStruct fil, int n, long vn_addr, long socket_addr)
{
  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, vn_addr, socket_addr));
}

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

main(int argc, char **argv)
{
  int i;
  long nfile, vn_addr, socket_addr, addr;
  char *error, *ptr;
  DataStruct fil;

  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(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);
  }
  print("Addr        Type  Ref  Msg    Fileops        Islot        Cred Offset"
        "    Flags");
  if(argc == 0){
    for(i=0;i<nfile;i++){
      if(!prfiles(fil, i, vn_addr, socket_addr)) quit(1);
```

Example 4-4: (continued)

```
    }
  }
  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, vn_addr, socket_addr)) quit(1);
      argv++;
    }
  }
  quit(0);
}
```

Example 4-5: Extension that uses global symbols: sum.c

```
#include <stdio.h>
#include "krash.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;

  check_args(argc, argv, help_string);
  read_var("utsname.nodename", STRING, &resp);
  sprintf(buf, "Hostname : %s", resp);
  print(buf);
  free(resp);
  read_var("cpu_avail", NUMBER, &avail);
```

Example 4-5: (continued)

```
  read_var("cpup.system_string", 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);
}
```

4.4 Build and Compile Considerations

After you have written the extension, you need to compile it. To do this, you need to access the `krashlib.o` and `array.o` object files and the header files in the `./src/usr/bin/kdbx/extensions` directory. The steps for building and compiling the extension `test` are as follows:

1. Create the `Makefile` file. Example 4-6 shows a sample `Makefile` that you can use.

Example 4-6: Sample Makefile

```
# Sample Makefile

ROOT = /usr/source_kit      # the root of the source tree
SRCDIR = $(ROOT)/src
OBJDIR = $(ROOT)/obj/alpha
EXT_DIR = $(OBJDIR)/usr/bin/kdbx/extensions
EXT_OBJS = $(EXT_DIR)/krashlib.o $(EXT_DIR)/array.o
CFLAGS = -I$(SRCDIR)/usr/bin/kdbx -I$(SRCDIR)/usr/bin/kdbx/extensions

all : file                  # Name of the extension

file : file.o $(EXT_OBJS)
        $(CC) -o file file.o $(EXT_OBJS)

file.o : file.c
```

2. Compile and link against the `krashlib.o` library file as follows:

```
% make test
```

3. Debug the extension, using `kdbx` and `dbx` together if necessary. See Section 4.5 for information on debugging your extension.

4. Place the custom extension in a directory that is accessible to other users. DEC OSF/1 extensions are located in the `/var/kdbx` directory.

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

(kdbx) test
Hostname : decosf.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)
```

4.5 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 a window dedicated to each debugger, though you can debug a task from a terminal.

If you are using a workstation or have two terminals, perform the following steps:

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:

   Session 1:

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

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

   Session 2:

```
# 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, issue the `procpd` alias to create the files `/tmp/pipein` and

/tmp/pipeout as follows:

Session 1:

```
(kdbx) procpd
```

The file pipein takes output from the dbx session and directs it as input to the kdbx session. The file pipeout takes output from the kdbx session and directs it as input to the dbx session.

In the dbx session, issue 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:

Session 2:

```
(dbx) run [ args ] < /tmp/pipeout > /tmp/pipein
```

3. As you step through the extension in the dbx session, you will 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, do the following:

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 will see the results of any action in the kdbx session. At this point, you can use the available dbx commands and options. For more information on the dbx debugger, see the *Programmer's Guide*.

Using the System Exercisers 5

This chapter describes how to use system exercisers to troubleshoot your DEC OSF/1 operating system. The exerciser commands reside in the `/usr/field` directory and allow you to test all or part of your system.

The system exercisers test the following areas:

- File systems – `fsx` (Section 5.3)
- System memory – `memx` (Section 5.4)
- Shared memory – `shmx` (Section 5.5)
- Peripherals – `diskx` (Section 5.6) and `tapex` (Section 5.7)
- Communications systems – `cmx` (Section 5.8)

In addition to the exercisers documented in this chapter, your system may also support the DEC Verifier and Exerciser Tool (VET), which provides a similar set of exercisers. VET is present on the installation kit as an optional subset. For information on VET, see the *DEC Verifier and Exerciser Tool User's Guide.*

5.1 Running System Exercisers

To run a system exerciser, you must be logged in as superuser and `/usr/field` must be your current directory.

The commands that invoke the system exercisers provide an option for specifying a file where diagnostic output is saved when the exerciser completes its task.

Most of the exerciser commands have an online help option that displays a description of how to use that exerciser. To access online help, use the –h option with a command. For example, to access help for the `diskx` exerciser, use the following command:

```
# diskx -h
```

The exercisers can be run in the foreground or the background and can be canceled at any time by pressing Ctrl/C in the foreground. You can run more than one exerciser at the same time; keep in mind, however, that the more processes you have running, the slower the system performs. Thus, before exercising the system extensively, make sure that no other users are on the

system.

There are some restrictions when you run a system exerciser over a Network File System (NFS) link or on a diskless system. For exercisers such as `fsx` that need to write to a file system, the target file system must be writable by root. Also, the directory from which an exerciser is executed must be writable by root because temporary files are written to the directory.

These restrictions can be difficult to adhere to because NFS file systems are often mounted in a way that prevents root from writing to them. Some of the restrictions may be adhered to by copying the exerciser into another directory and then executing it.

5.2 Using Exerciser Diagnostics

When an exerciser is halted (by either Ctrl/C or timing out), diagnostics are displayed and are stored in the exerciser's most recent log file. The diagnostics inform you of the test results.

Each time an exerciser is invoked, a new log file is created in the `/usr/field` directory. For example, when you execute the `fsx` command for the first time, a log file named `#LOG_FSX_01` is created. The log files contain records of each exerciser's results and consist of the starting and stopping times, and error and statistical information. The starting and stopping times are also logged into the default system error log file, `/var/adm/binary.errlog`. This file also contains information on errors reported by the device drivers or by the system.

The log files provide a record of the diagnostics. However, after reading a log file, you should delete it because an exerciser can have only nine log files. If you attempt to run an exerciser that has accumulated nine log files, the exerciser tells you to remove some of the old log files so that it can create a new one.

If an exerciser finds errors, you can determine which device or area of the system has the difficulty by looking at `/var/adm/binary.errlog`, using the `uerf` command. For information on the error logger, see the manual *System Administration*. For the meanings of the error numbers and signal numbers, see the `intro`(2) and `sigvec`(2) reference pages.

5.3 Exercising a File System

Use the `fsx` command to exercise the local file systems. The `fsx` command exercises the specified local file system by initiating multiple processes, each of which creates, writes, closes, opens, reads, validates, and unlinks a test file of random data. For more information, see the `fsx`(8) reference page.

Note

Do not test Network File System (NFS) file systems with the `fsx` command.

The `fsx` command has the following syntax:

fsx [**–h**] [**–o***file*] [**–t***min*] [**–f***path*] [**–p***num*]

You can specify one or more of the following options:

–h
: Displays the help message for the `fsx` command.

–o*file*
: Saves the output diagnostics in *file*.

–p*num*
: Specifies the number of `fsxr` processes you want `fsx` to initiate. The maximum number of processes is 250. The default is 20.

–f*path*
: Specifies the pathname of the file system directory you want to test. For example, –`f/usr` or –`f/mnt`. The default is `/usr/field`.

–t*min*
: Specifies how many minutes you want the `fsx` command to exercise the file system. If you do not specify the –`t` option, the `fsx` command runs until you terminate it by pressing Ctrl/C in the foreground.

The following example of the `fsx` command tests the `/usr` file system with five `fsxr` processes running for 60 minutes in the background:

```
# fsx -p5 -f/usr -t60 &
```

5.4 Exercising System Memory

Use the `memx` command to exercise the system memory. The `memx` command exercises the system memory by initiating multiple processes. By default, the size of each process is defined as the total system memory in bytes divided by 20. The minimum allowable number of bytes per process is 4095. The `memx` command runs 1s and 0s, 0s and 1s, and random data patterns in the allocated memory being tested. For more information, see the `memx`(8) reference page

The `memx` command is restricted by the amount of available swap space. The size of the swap space and the available internal memory determine how many processes can run simultaneously on your system. For example, if there are 16 MB of swap space and 16 MB of memory, all of the swap space

will be used if all 20 initiated processes (the default) run simultaneously. This would prevent execution of other process.

Therefore, on systems with large amounts of memory and small amounts of swap space, you must use the –p or –m option, or both, to restrict the number of memx processes or to restrict the size of the memory being tested.

The memx command has the following syntax:

memx –s [**–h**] [**–o***file*] [**–t***min*] [**–m***size*] [**–p***num*]

You can specify one or more of the following options:

–h
: Displays the help message for the memx command.

–o*file*
: Saves the output diagnostics in *file*.

–m*size*
: Specifies the amount of memory in bytes for each process you want to test. The default is the total amount of memory divided by 20, with a minimum size of 4095 bytes.

–p*num*
: Specifies the number of memxr processes to initiate. The maximum number is 20, which is also the default.

–s
: Disables the automatic invocation of the shared memory exerciser, shmx.

–t*min*
: Specifies how many minutes you want the memx command to exercise the memory. If you do not specify the –t option, the memx command runs until you terminate it by pressing Ctrl/C in the foreground.

The following example of the memx command initiates five memxr processes that test 4095 bytes of memory and runs in the background for 60 minutes:

```
# memx –m4095 –p5 –t60 &
```

5.5 Exercising Shared Memory

Use the shmx command to exercise the shared memory segments. The shmx command spawns a background process called shmxb. The shmx command writes and reads the shmxb data in the segments, and the shmxb process writes and reads the shmx data in the segments.

Using shmx, you can test the number and the size of memory segments and shmxb processes. The shmx exerciser runs until the process is killed or until the time specified by the –t option is exhausted.

You automatically invoke the shmx exerciser when you start the memx exerciser, unless you specify the memx command with the –s option. You can also invoke the shmx exerciser manually. The shmx command has the following syntax:

/usr/field/shmx [**–h**] [**–o***file*] [**–v**] [**–t***time*] [**–m***size*] [**–s***n*]

The shmx command options are as follows:

-h
: Prints the help message for the shmx command.

–o*file*
: Saves diagnostic output in *file*.

–v
: Uses the fork system call instead of the vfork system call to spawn the shmxb process.

-t*time*
: Specifies *time* as the run time in minutes. The default is to run until the process is killed.

-m*size*
: Specifies *size* as the memory segment size, in bytes, to be tested by the processes. The *size* value must be greater than zero. The default is the value of the SHMMAX and SHMSEG system parameters, which are set in the /sys/include/sys/param.h file.

-s*n*
: Specifies *n* as the number of memory segments. The default (and maximum) number of segments is 3.

The following example tests the default number of memory segments, each with a default segment size:

```
# shmx &
```

The following example runs three memory segments of 100,000 bytes for 180 minutes:

```
# shmx -t180 -m100000 -s3 &
```

5.6 Exercising a Disk Drive

Use the `diskx` command to exercise the disk drives. The main functional areas that are tested include the following:

- Reads, writes, and seeks
- Performance
- Disktab entry verification

Some of the tests involve writing to the disk; for this reason, use the exerciser cautiously on disks that contain useful data that the exerciser could overwrite. Tests that write to the disk first check for the existence of file systems on the test partitions and partitions that overlap the test partitions. If a file system is found on these partitions, you are prompted to determine if testing should continue.

You can use the `diskx` command options to specify the tests that you want performed and to specify the parameters for the tests. For more information, see the `diskx`(8) reference page.

The `diskx` command has the following syntax:

diskx [*options*] [*parameters*] **–f** *devname*

The `–f` *devname* option specifies the device special file on which to perform testing. The *devname* variable specifies the name of the block or character special file that represents the disk to be tested. The file name must begin with an `r` (for example, `rz1`). The last character of the file name can specify the disk partition to test.

If a partition is not specified, all partitions are tested. For example, if the *devname* variable is `/dev/rra0`, all partitions are tested. If the *devname* variable is `/dev/rra0a`, the `a` partition is tested. This parameter must be specified and can be used with all test options.

The following options specify the tests to be run on disk:

`–d`

Tests the disk's `disktab` file entry. The `disktab` entry is obtained by using the `getdiskbyname` library routine. This test only works if the specified disk is a character special file. See the `disktab`(4) reference page for more information.

`–h`

Displays a help message describing test options and parameters.

`–p`

Specifies a performance test. Read and write transfers are timed to measure device throughput. Data validation is not performed as part of this test. Testing uses a range of transfer sizes if the `–F` option is not

specified.

The range of transfer sizes is divided by the number specified with the `perf_splits` parameter to obtain a transfer size increment. For example, if the `perf_splits` parameter is set to 10, tests are run starting with the minimum transfer size and increasing the transfer size by 1/10th of the range of values for each test repetition. The last transfer size is set to the specified maximum transfer size.

If you do not specify a number of transfers, the transfer count is set to allow the entire partition to be read or written. In this case, the transfer count varies, depending on the transfer size and the partition size.

The performance test runs until completed or until interrupted; the time is not limited by the `–minutes` parameter. This test can take a long time to complete, depending on the test parameters.

In order to acheive maximum throughput, specify the `–S` option to cause sequential transfers. If the `–S` option is not specified, transfers are done to random locations. This may slow down the observed throughput because of associated head seeks on the device.

`–r`

Specifies a read-only test. This test reads from the specified partitions. Specify the `–n` option to run this test on the block special file.

This test is useful for generating system I/O activity. Because it is a read-only test, you can run more than one instance of the exerciser on the same disk.

`–w`

Specifies a write test. This test verifies that data can be written to the disk and can be read back to verify the data. Seeks are also done as part of this test. This test provides the most comprehensive coverage of disk transfer functions because it uses reads, writes, and seeks. This test also combines sequential and random access patterns.

This test performs the following operations using a range of transfer sizes; a single transfer size is utilized if the `–F` attribute is specified:

- Sequentially writes the entire test partition, unless the number of transfers has been specified using the `–num_xfer` parameter
- Sequentially reads the test partition

The data read from the disk is examined to verify it. Then, if random transfer testing has not been disabled (using the `–S` attribute), writes are issued to random locations on the partition. After the random writes are completed, reads are issued to random locations on the partition. The data read from random locations is examined to verify it.

The following options modify the behavior of the test:

–F

Performs fixed size transfers. If this option is not specified, transfers are done using random sizes. This option can be used with the –p, –r, and –w test options.

–i

Specifies interactive mode. In this mode, you are prompted for various test parameters. Typical parameters include the transfer size and the number of transfers. The following scaling factors are allowed:

- k or K (for kilobyte (1024 * n))
- b or B (block (512 * n))
- m or M (megabyte (1024 * 1024 * n))

For example 10K would specify 10,240 bytes.

–Q

Suppresses performance analysis of read transfers. This option only performs write performance testing. To perform only read testing and to skip the write performance tests, specify the –R option. The –Q option can be used with the –p test option.

–R

Opens the disk in read-only mode. This option can be used with all test options.

–S

Performs transfers to sequential disk locations. If this option is not specified, transfers are done to random disk locations. This option can be used with the –p, –r, and –w test options.

–T

Directs output to the terminal. This option is useful if output is directed to a log file by using the –o option. If you specify the –T option after the –o option, output is directed to both the terminal and the log file. The –T option can be used with all test options.

–Y

Does not prompt you to confirm that you want to continue the test if file systems are found when the disk is examined; testing proceeds.

In addition to the options, you can also specify test parameters. You can specify test parameters on the **diskx** command line or interactively with the –i option. If you do not specify test parameters, default values are used.

To use a parameter, specify the parameter name, a space, and the numeric

value. For example, you could specify the following *parameter*:

```
-perf_min 512
```

You can use the following scaling factors:

- k or K (for kilobyte (1024 * n))
- b or B (for block (512 * n))
- m or M (for megabyte (1024 * 1024 * n))

For example, 10K would specify 10,240 bytes.

For example, `-perf_min 10K` causes transfers to be done in sizes of 10,240 bytes.

You can specify one or more of the following *parameters:*

`-debug`
: Specifies the level of diagnostic output to be produced. The greater the number specified, the more output is produced describing the exerciser operations. This parameter can be used with all test options.

`-err_lines`
: Specifies the maximum number of error messages that are produced as a result of an individual test. A limit on error output prevents a large number of diagnostic messages if persistent errors occur. This parameter can be used with all test options.

`-minutes`
: Specifies the number of minutes to test. This parameter can be used with the `-r` and `-w` test options.

`-max_xfer`
: Specifies the maximum transfer size to be performed. If transfers are done using random sizes, the sizes are within the range specified by the `-max_xfer` and `-min_xfer` parameters. If fixed size transfers are specified (see the `-F` option), transfers are done in a size specified by the `-min_xfer` parameter.

  Specify transfer sizes to the character special file in multiples of 512 bytes. If the specified transfer size is not an even multiple, the value is rounded down to the nearest 512 bytes. This parameter can be used with the `-r` and `-w` test options.

`-min_xfer`
: Specifies the minimum transfer size to be performed. This parameter can be used with the `-r` and `-w` test options.

`-num_xfer`
: Specifies the number of transfers to perform before changing the partition that is currently being tested. This parameter is only useful if

more than one partition is being tested. If this parameter is not specified, the number of transfers is set to a number that completely covers a partition. This parameter can be used with the –r and –w test options.

–o*filename*
: Sends output to the specified file name. The default is to display output on the terminal screen. This parameter can be used with all test options.

–perf_max
: Specifies the maximum transfer size to be performed. If transfers are done using random sizes, the sizes are within the range specified by the –perf_min and –perf_max parameters. If fixed size transfers are specified (see the –F option), transfers are done in a size specified by the –perf_min parameter. This parameter can be used with the –p test option.

–perf_min
: Specifies the minimum transfer size to be performed. This parameter can be used with the –p test option.

–perf_splits
: Specifies how the transfer size will change if you test a range of transfer sizes. The range of transfer sizes is divided by the number specified with the perf_splits parameter to obtain a transfer size increment. For example, if the perf_splits parameter is set to 10, tests are run starting with the minimum transfer size and increasing the transfer size by 1/10th of the range of values for each test repetition. The last transfer size is set to the specified maximum transfer size. This parameter can be used with the –p test option.

–perf_xfers
: Specifies the number of transfers to be performed in performance analysis. If this value is not specified, the number of transfers is set equal to the number that is required to read the entire partition. This parameter can be used with the –p test option.

The following example performs read-only testing on the character device special file that /dev/rrz0 represents. Because a partition is not specified, the test reads from all partitions. The default range of transfer sizes is used. Output from the exerciser program is displayed on the terminal screen.

```
# diskx -f /dev/rrz0 -r
```

The following example runs on the a partition of /dev/rz0. Program output will be logged to the diskx.out file. The program output level is

set to 10 and causes additional output to be generated.

```
# diskx -f /dev/rz0a -o diskx.out -d -debug 10
```

The following example shows that performance tests are run on the `a` partition of `/dev/rz0`. Program output is logged to the `diskx.out` file. The `–S` option causes sequential transfers for the best test results. Testing is done over the default range of transfer sizes.

```
# diskx -f /dev/rz0a -o diskx.out -p -S
```

The following command runs the read test on all partitions of the specified disks. The disk exerciser is invoked as three separate processes, which generate extensive system I/O activity. The command shown in this example can be used to test system stress.

```
# diskx -f /dev/rrz0 -r &; diskx -f /dev/rrz1 -r &; diskx -f
/dev/rrz2 -r &
```

5.7 Exercising a Tape Drive

Use the `tapex` command to exercise a tape drive. The `tapex` command writes, reads, and validates random data on a tape device from the beginning of the tape (BOT) to the end of the tape (EOT). The `tapex` command also performs positioning tests for records and files, and tape transportability tests. For more information, refer to the `tapex`(8) reference page.

Some `tapex` options perform specific tests (for example, an end-of-media test). Other options modify the tests, for example, by enabling caching.

The `tapex` command has the following syntax:

tapex [*options*] [*parameters*]

You can specify one or more of the *`options`* described in Table 5-1. In addition to *`options`*, you can also specify test *`parameters`*. You specify parameters on the `tapex` command line or interactively with the `–i` option. If you do not specify test parameters, default values are used.

To use a test parameter, specify the parameter name, a space, and the number value. For example, you could specify the following *`parameter`*:

```
–min_rs 512
```

Note that you can use the following scaling factors:

- k or K (for kilobyte (1024 * n))
- b or B (for block (512 * n))
- m or M (for megabyte (1024 * 1024 * n))

For example, 10K would specify 10240 bytes.

The following parameters can be used with all tests:

`-err_lines`
Specifies the error printout limit.

`-fixed` *bs*
Specifies a fixed block device. Record sizes for most devices default to multiples of the blocking factor of the fixed block device as specified by the *bs* argument.

The following parameters can be used with the `-a` option, which measures performance:

`-perf_num`
Specifies the number of records to write and read.

`-perf_rs`
Specifies the size of records.

Other parameters are restricted for use with specific `tapex` options. Option-specific parameters are documented in Table 5-1.

Table 5-1: tapex Options and Option Parameters

tapex Option	Option and Parameter Descriptions

`-a`	Specifies the performance measurement test, which calculates the tape transfer bandwidth for writes and reads to the tape by timing data transfers.
	The following parameters can be used with the `-a` option:
	`-perf_num`
	Specifies the number of records to write and read.
	`-perf_rs`
	Specifies the size of records.

`-b`	Causes the write/read tests to run continuously until the process is killed. This option can be used with the `-r` and `-g` options.
`-c`	Enables caching on the device, if supported. This option does not specifically test caching; it enables the use of caching on a tape device while other tests are running.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
–C	Disables caching on TMSCP tape devices. If the tape device is a TMSCP unit, then caching is the default mode of test operation. This option causes the tests to run in noncaching mode.

–d	Tests the ability to append records to the media. First, the test writes records to the tape. Then, it repositions itself back one record and appends additional records. Finally, the test does a read verification. This test simulates the behavior of the `tar –r` command.
	
	The following parameters can be used with the –d option:
	
	`–no_overwrite`
	Prevents the append-to-media test from being performed on tape devices that do not support this test. Usually, you use this parameter with the –E option.
	
	`–tar_num`
	Specifies the number of additional and appended records.
	
	`–tar_size`
	Specifies the record size for all records written in this test.

–e	Specifies end-of-media (EOM) test. First, this test writes data to fill a tape; this action can take a long time for long tapes. It then performs some reads and writes past the end of the media; these actions should fail. Finally, it enables writing past the end of the media, writes to the tape, and reads back the records for validation purposes.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
	The following parameters can be used with the –e option: –end_num Specifies the number or records to be written past EOM. (Note that specifying too much data to be written past EOM can cause a reel-to-reel tape to go off line): –end_rs Specifies the record size.
–E	Runs an extensive series of tests in sequential order. Depending on tape type and CPU type, this series of tests can take up to 10 hours to complete.
–f /dev/rmt #?	Specifies the name of the device special file that corresponds to the tape unit being tested. The number sign variable (#) specifies the unit number. The question mark variable (?) specifies the letter h for the high density device or l for the low density device. The default tape device is /dev/rmt0h.
–F	Specifies the file-positioning tests. First, files are written to the tape and verified. Next, every other file on the tape is read. Then, the previously unread files are read by traversing the tape backwards. Finally, random numbers are generated, the tape is positioned to those locations, and the data is verified. Each file uses a different record size. The following parameters can be used with the –F option: –num_fi Specifies the number of files. –pos_ra Specifies the number of random repositions. –pos_rs Specifies the record size. –rec_fi Specifies the number of records per file.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
–G	Specifies the file-positioning tests on a tape containing data. This option can be used with the –F option to run the file position tests on a tape that has been written to by a previous invocation of the –F test. To perform this test, you must use the same test parameters (for example, record size and number of files) that you used when you invoked the –F test to write to the tape. No other data should have been written to the tape since the previous –F test.
–g	Specifies random record size tests. This test writes records of random sizes. It reads in the tape, specifying a large read size; however, only the amount of data in the randomly sized record should be returned. This test only checks return values; it does not validate record contents. The following parameter is used with the –g option: –rand_num Specifies the number of records to write and read.
–h	Displays a help message describing the tape exerciser.
–i	Specifies interactive mode. In this mode, you are prompted for various test parameters. Typical parameters include the record size and the number of records to write. The following scaling factors are allowed: • k or K (for kilobyte (1024 * n)) • b or B (for block (512 * n)) • m or M (for megabyte (1024 * 1024 * n)) For example, 10K would specify 10,240 bytes.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
–j	Specifies the write phase of the tape-transportability tests. This test writes a number of files to the tape and then verifies the tape. After the tape has been successfully verified, it is brought off line, moved to another tape unit, and read in with the –k option. This test proves that a tape can be written to on one drive and read from on another drive. The –j option is used with the –k option. Note that the parameters used with –j option must be the same parameters used with the –k option. The following parameters can be used with the –j and –k options: –tran_file Specifies the number of files to write or read. –tran_rec Specifies the number of records contained in each file. –tran_rs Specifies the size of each record.
–k	Specifies the read phase of the tape-transportability tests. This test reads a tape that was written by the –j test and verifies that the expected data is read from the tape. This test proves that a tape can be written to on one drive and read from on another. As stated in the description of the –j option, any parameters specified with the –j option must be specified with the –k option. (See the description of the –j option for information on the parameters that apply to the –j and –k options.)
–L	Specifies the media loader test. For sequential stack loaders, the media is loaded, written to, and verified. Then, the media is unloaded, and the test is run on the next piece of media. This verifies that all of the media in the input deck can be written to. To run this test in read-only mode, also specify the –w option.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
–l	Specifies the end-of-file (EOF) test. This test verifies that a 0 (zero) byte count is returned when a tape mark is read and that an additional read fetches the first record of the next tape file.
–m	Displays tape contents. This is not a test. This option reads the tape sequentially and prints out the number of files on the tape, the number of records in each file, and the size of the records within the file. The contents of the tape records are not examined.
–o *filename*	Sends output to the specified file name. The default sends output to the terminal screen.
–p	Runs both the record-positioning and file-positioning tests. For more information, refer to descriptions of the –R and –F options.
–q	Specifies the command timeout test. This test verifies that the driver allows enough time for completion of long operations. This test writes files to fill the tape. It then performs a rewind, followed by a forward skip to the last file. This test is successful if the forward skip operation is completed without error.
–r	Specifies the record size test. A number of records are written to the tape and then verified. This process is repeated over a range of record sizes. The following parameters can be used with the –r option: –inc Specifies the record increment factor. –max_rs Specifies the maximum record size. –min_rs Specifies the minimum record size. –num_rec Specifies the number of records. –t Specifies a time limit (in minutes). The default is to run the test until it is complete.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
–R	Specifies the record-positioning test. First, records are written to the tape and verified. Next, every other record on the tape is read. Then, the other records are read by traversing the tape backwards. Finally, random numbers are generated; the tape is positioned to those locations, and the data is verified. The following parameters can be used with the -R option: –pos_num Specifies the number of records. –pos_ra Specifies the number of random repositions. –pos_rs Specifies the record size.
–s	Specifies the record size behavior test. Verifies that a record that is read returns one record (at most) or the read size, whichever is less. The following parameters can be used with the –s option: –num_rec Specifies the number of records. –size_rec Specifies the record size.
–S	Specifies single record size test. This test modifies the record size test (the –r option) to use a single record size. The following parameters can be used with the -S option: –inc Specifies the record increment factor. –max_rs Specifies the maximum record size. –min_rs Specifies the minimum record size.

Table 5-1: (continued)

tapex Option	Option and Parameter Descriptions
	–num_rec Specifies the number of records.
–T	Displays output to the terminal screen. This option is useful if you want to log output to a file with the –o option and also have the output displayed on your terminal screen. This option must be specified after the –o option in the command line.
–v	Specifies verbose mode. This option causes detailed information to be output. For example, it lists the operations the exerciser is performing (such as record counts), and detailed error information. Information provided by this option can be useful for debugging purposes.
–V	Specifies enhanced verbose mode. This option causes more detailed information than the –v option to be output. The additional output consists of status information on exerciser operations. Information provided by this option can be useful for debugging purposes.
–w	Opens the tape as read-only. This mode is useful only for tests that do not write to the media. For example, it allows the –m test to be run on a write-protected media.
–Z	Initializes the read buffer to the nonzero value 0130. This can be useful for debugging purposes. If the –Z option is not specified, all elements of the read buffer are initialized to zero. Many of the tests first initialize their read buffer and then perform the read operation. After reading a record from the tape, some tests validate that the unused portions of the read buffer remain at the value to which they were initialized. For debugging purposes, you can set this initialized value to a number other than zero. In this case, you can use the arbitrary value 0130.

The following example runs an extensive series of tests on tape device

rmt1h and sends all output to the tapex.out file:

```
# tapex –f /dev/rmt1h -E -o tapex.out
```

The following example performs random record size tests and outputs information in verbose mode. This test runs on the default tape device /dev/rmt0h, and the output is sent to the terminal screen.

```
# tapex -g -v
```

The following example performs read and write record testing using record sizes in the range 10K to 20K. This test runs on the default tape device /dev/rmt0h, and the output is sent to the terminal screen.

```
# tapex -r -min_rs 10k -max_rs 20k
```

The following example performs a series of tests on tape device /dev/rmt0h, which is treated as fixed block device in which record sizes for tests are multiples of the blocking factor 512 kilobytes. The append-to-media test is not performed.

```
# tapex -f /dev/rmt0h -fixed 512 -no_overwrite
```

5.8 Exercising the Terminal Communication System

Use the cmx command to exercise the terminal communications system. The cmx command writes, reads, and validates random data and packet lengths on the specified communications lines.

The lines you exercise must have a loopback connector attached to the distribution panel or the cable. Also, the line must be disabled in the /etc/inittab file and in a nonmodem line; that is, the CLOCAL flag must be set to on. Otherwise, the cmx command repeatedly displays error messages on the terminal screen until its time expires or until you press Ctrl/C. For more information, refer to the cmx(8) reference page.

You cannot test pseudodevice lines or lta device lines. Pseudodevices have p, q, r, s, t, u, v, w, x, y, or z as the first character after tty, for example, ttyp3.

The cmx command has the following syntax:

/usr/field/cmx [**–h**] [**–o***file*] [**–t***min*] **–l** *line*

The cmx command options are as follows:

–h
: Prints a help message for the cmx command.

–o*file*
: Saves output diagnostics in *file*.

–t*min*

Specifies how many minutes you want the cmx command to exercise the communications system. If you do not specify the –t option, the cmx command runs until you terminate it by pressing Ctrl/C in the foreground.

–l *line*

Specifies the line or lines you want to test. The possible values for *line* are found in the /dev directory and are the last two characters of the tty device name. For example, if you want to test the communications system for devices named tty02, tty03, and tty14, specify 02, 03, and 14, separated by spaces, for the *line* variable. In addition, the line variable can specify a range of lines to test. For example, 00-08.

The following example exercises communications lines tty22 and tty34 for 45 minutes in the background:

```
# cmx -l 22 34 -t45 &
```

The following example exercises lines tty00 through tty07 until you press Ctrl/C:

```
# cmx -l 00-07
```

Index

G

H

I

K

L

M

N

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[a]	————	SSB Order Processing - NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Kernel Debugging
AA-PS2TB-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

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 _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ----------

Cut Along Dotted Line

Reader's Comments

DEC OSF/1
Kernel Debugging
AA-PS2TB-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

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 _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut
Along
Dotte
Line