Technical Note TN2063
Understanding and Debugging Kernel Panics

CONTENTS

When the kernel crashes on Mac OS X, the system displays a panic message. At this point the system will have to be restarted. But before hitting the reset button, how can one find out what caused the crash?

This technote addresses kernel panics: what they are and how to debug the code that caused the panic.

The foundation of Mac OS X is a core operating system commonly known as Darwin. Although the Darwin kernel runs on both PowerPC and Intel x86 architectures, this technote discusses PowerPC exception handling only.

This technote contains links to source files available from the Darwin repository. Access to these files requires a username and password obtained by agreeing to the Apple Public Source License.

[Nov 11 2002]

What is a Kernel Panic?

In UNIX, a **panic** is an unrecoverable system error detected by the kernel [1] as opposed to similar errors detected by user space code. It is possible for kernel code to indicate such a condition by calling the `panic` function located in the header file `sys/systm.h`. However, most panics are the result of unhandled processor exceptions in kernel code, such as references to invalid memory addresses. These are typically indicative of a bug somewhere in the call chain leading up to the panic.

Back to top

Basics of PowerPC Exception Handling in Mac OS X's Darwin Kernel

An **exception** is a condition encountered by the processor that requires special processing. The PowerPC microprocessor family handles exceptions by switching to supervisor state, saving the processor state to certain registers, and then jumping to an exception handler routine. Each major type of exception (data memory access, alignment, etc.) has its own exception vector located at an absolute address defined in the PowerPC architecture.

The most common exceptions are:

- **DSI** (data storage interrupt, or data memory access) exceptions, caused by an attempt to access data at an invalid memory address, such as dereferencing a NULL pointer.

- **ISI** (instruction storage interrupt) exceptions, caused by an attempt to execute an instruction at an invalid memory address, such as branching to location zero.
- Illegal instruction exceptions, caused by an attempt to execute an instruction with an invalid opcode.

Details on PowerPC exception handling can be found in Chapter 6 of the book *PowerPC Microprocessor Family: The Programming Environments* (hereafter referred to as *TPE*).

Several processor registers that are involved in exception handling are displayed when a panic is caused by an unhandled exception. These registers are:

DSISR

Identifies the cause of DSI and alignment exceptions such as a direct-store error exception, or the operand of an integer double-word load or store instruction is not word-aligned.

Data Access Register (DAR)

Contains the effective address of the memory element which caused a DSI or alignment exception.

Machine State Register (MSR)

Defines the state of the processor. Settings include interrupt enable, privilege level, machine check enable, and address translation bits.

Machine Status Save/Restore Register 0 (SRR0)

Contains the address used to calculate where instruction processing should continue after the exception is handled. Depending on the exception, this may be the effective address of the instruction which caused the exception or the next instruction in the program flow. This register is displayed in panics as **PC** (program counter).

Machine Status Save/Restore Register 1 (SRR1)

Contains exception-specific information and selected bits from the MSR at the time the exception occurred. This register is displayed in panics as **MSR**.

Link Register (LR)

Contains the address of the instruction following the last subroutine call (`bl`: branch then link) instruction.

General Purpose Register 1 (GPR1)

Used as the stack pointer to store parameters and other temporary data items. This register is displayed in panics as **R1**.

Details on the PowerPC register set can be found in *TPE* Chapter 2.

The Darwin kernel follows this execution flow when handling a PowerPC exception:

`xnu/osfmk/ppc/lowmem_vectors.s: L_handlerXXXX`

(where `XXXX` is the exception handler vector in the range 100 to 2FFF; only 100-2000 are currently used)

`xnu/osfmk/ppc/lowmem_vectors.s: L_exception_entry`

`xnu/osfmk/ppc/hw_exception.s: thandler`

`xnu/osfmk/ppc/trap.c: trap`

`xnu/osfmk/ppc/trap.c: unresolved_kernel_trap`

The last function (`unresolved_kernel_trap`) is where panic information is displayed.

Back to top

What Does a Panic Look Like?

Listing 1 is a typical panic display from a Mac OS X 10.2.1 system. Line numbers have been added for ease of reference.

Listing 1. Example panic dump.

```
1 Unresolved kernel trap(cpu 0): 0x300 - Data access DAR=0xdeadbeef PC=0x0e692550
2 Latest crash info for cpu 0:
3    Exception state (sv=0x0EB5DA00)
4       PC=0x0E692550; MSR=0x00009030; DAR=0xDEADBEEF; DSISR=0x42000000; LR=0x0E692530;
        R1=0x081DBC20; XCP=0x0000000C (0x300 - Data access)
5       Backtrace:
6          0x0E6924A8 0x00213A88 0x00213884 0x002141D4 0x00214830
           0x00204CB0 0x00204C74
7       Kernel loadable modules in backtrace (with dependencies):
8          com.apple.dts.driver.PanicDriver(1.0)@0xe691000
9             dependency: com.apple.iokit.IOUSBFamily(1.9.2)@0xed9c000
10 Proceeding back via exception chain:
11    Exception state (sv=0x0EB5DA00)
12       previously dumped as "Latest" state. skipping...
13    Exception state (sv=0x0EB64A00)
14       PC=0x00000000; MSR=0x0000D030; DAR=0x00000000;
                 DSISR=0x00000000; LR=0x00000000; R1=0x00000000; XCP=0x00000000 (Unknown)
15
16 Kernel version:
17 Darwin Kernel Version 6.1:
18 Fri Sep  6 23:24:34 PDT 2002; root:xnu/xnu-344.2.obj~2/RELEASE_PPC
19
20
21 Memory access exception (1,0,0)
22 ethernet MAC address: 00:0a:11:22:33:44
23 ip address: 169.254.180.203
24
25 Waiting for remote debugger connection.
```

Starting with Mac OS X 10.2, a panic is indicated by the multi-lingual alert shown in Figure 1. After restarting the system, a file called `panic.log` should be present in `/Library/Logs`. This file contains the same data as the panic dump on the screen.

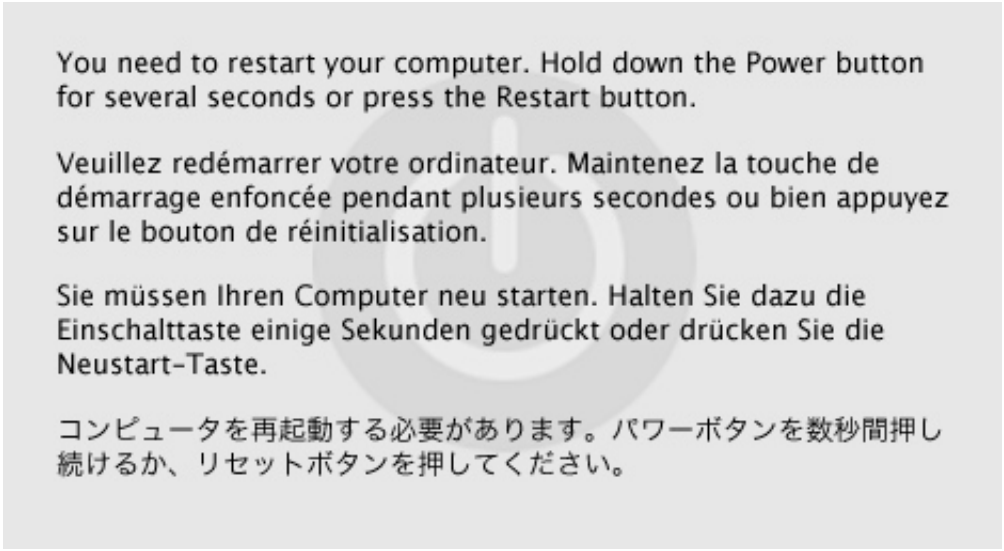

Figure 1. Mac OS X 10.2 panic alert.

If remote debugging has been enabled via the `debug` parameter in `boot-args`, once either the panic alert or textual panic dump is displayed, the system is waiting for a connection from a remote GDB debugger session. For more details on remote (two-machine) debugging, please refer to the Hello Debugger tutorial.

Note:
A panic log file is **not** written if remote debugging has been enabled.

A list of flags affecting remote debugging is in Table 19-1 of *Inside Mac OS X: Kernel Programming*.

Back to top

How to Read the Panic Display

For each line of the panic display, the name of the kernel source file and function that displays that line is given, followed by an explanation of the information on that line.

Line 1 `xnu/osfmk/ppc/trap.c: unresolved_kernel_trap`

`Unresolved kernel trap`: Textual description of the cause of the panic. This is the parameter passed to the `panic` function.

`(cpu 0)`: The number of the CPU on which the exception has occurred. Useful on multiprocessor systems. Note that on a multiprocessor system it's possible for one processor to be panicked while the other continues to run.

`0x300 - Data access`: trap name. This is a textual description of the exception. The trap names are found in the array `trap_type` in `xnu/osfmk/ppc/trap.c`. The hardware exception code `trapno` (defined in `xnu/osfmk/ppc/exception.h`) is initially set by the exception handler `xnu/osfmk/ppc/lowmem_vectors.s: L_handlerXXXX` where `XXXX` is the PowerPC exception vector. The index into the `trap_type` array is computed by dividing `trapno` by `T_VECTOR_SIZE`, defined to be 4 (the size of a function pointer), also in `xnu/osfmk/ppc/exception.h`.

The `0x300` at the beginning is the PowerPC exception vector. Using this value one can look up details on the specific exception in *TPE* Chapter 6.

`DAR`: contents of Data Access Register
`PC`: contents of register SRR0

The interpretation of DAR and PC varies depending on the definition of each exception.

Back to Listing 1

Line 2 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 3 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Exception states are stored in data structures of type `savearea` (see `xnu/osfmk/ppc/exception.h`). `sv` is the address of the `savearea` for the latest exception.

Back to Listing 1

Line 4 `xnu/osfmk/ppc/model_dep.c: dump_savearea`

`PC`: contents of SRR0
`MSR`: contents of SRR1
`DAR`: contents of Data Access Register
`DSISR`: contents of DSISR
`LR`: contents of Link Register
`R1`: contents of GPR1
`XCP`: This is not a register but is the exception code stored in the `savearea` corresponding to the current exception. It is followed by the trap name (see line 1).

Back to Listing 1

Line 5 `xnu/osfmk/ppc/model_dep.c: dump_backtrace`

Back to Listing 1

Line 6 `xnu/osfmk/ppc/model_dep.c: dump_backtrace`

This is the actual stack backtrace. The initial stack pointer is the value of GPR1 in the `savearea`. The value of LR from the linkage area of the stack frame is printed, then the next stack frame is located using the value of GPR1 saved in the stack frame. Up to 32 stack frames will be printed, fewer than that if a zero GPR1 is encountered or if a `savearea` exists for an earlier exception.

Details on the PowerPC stack as used on Mac OS X can be found in the section "Power PC Stack Structure" of the book Inside Mac OS X: Mach-O Runtime Architecture.

The backtrace is typically the most useful information in a panic dump because it can be used to reconstruct the call chain that led to the exception. This is discussed in the next section "Isolating the Crash."

Back to Listing 1

Line 7 `xnu/osfmk/kern/kmod.c: kmod_dump`

This looks at the addresses in the backtrace and prints out the module name, version, and starting address of each kernel loadable module in the backtrace. (A kernel loadable module is simply the executable portion of a kernel extension, or kext.) It also prints out the same information for the dependencies of each kernel extension. The module name and version is the same as that shown by the `kextstat` command (`kmodstat` prior to Mac OS X 10.2) and is the value of `MODULE_NAME` and `MODULE_VERSION` in the Project Builder build settings. The dependencies are those specified in the `OSBundleLibraries` property in the Project Builder bundle settings.

Back to Listing 1

Line 8 `xnu/osfmk/kern/kmod.c: kmod_dump`

Back to Listing 1

Line 9 `xnu/osfmk/kern/kmod.c: kmod_dump`

Back to Listing 1

Line 10 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Each exception state is now dumped. The first one was already shown in lines 3 through 6 (note the same value of `sv` in both locations) so is skipped.

Back to Listing 1

Line 11 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 12 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 13 `xnu/osfmk/ppc/model_dep.c: dump_savearea`

Same as line 3.

Back to Listing 1

Line 14 `xnu/osfmk/ppc/model_dep.c: dump_savearea`

Same as line 4.

Back to Listing 1

Line 15 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 16 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

This prints the value of the kernel global variable `version`, set at build time. The value contains embedded newline characters, so it wraps from line 17 to line 19.

The string "Fri Sep 6 23:24:34 PDT 2002" is the date and time the kernel was built. The string "xnu/xnu-344.2.obj~2/RELEASE_PPC" is the object directory in which the kernel was built. The "xnu-344.2" part contains the same version number as the CVS tag for the Darwin source revision used to build this kernel (in this case Apple-344-2). This would allow one to build a custom kernel for those cases where source debugging of the kernel itself was desired.

To see the version of a running kernel, use the `sysctl` command as illustrated in Listing 2.

Listing 2. Displaying the kernel version

```
[localhost:~] me% sysctl kern.version
kern.version = Darwin Kernel Version 6.1:
Fri Sep  6 23:24:34 PDT 2002; root:xnu/xnu-344.2.obj~2/RELEASE_PPC


[localhost:~] me%
```

The steps to build a custom kernel can be found in the chapter "Building and Debugging Kernels" of the book Inside Mac OS X: Kernel Programming.

Back to Listing 1

Line 17 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 18 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 19 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

Line 20 `xnu/osfmk/ppc/model_dep.c: print_backtrace`

Back to Listing 1

The next three calls do not produce any output:

```
xnu/osfmk/ppc/misc_asm.s: Call_Debugger
xnu/osfmk/ppc/model_dep.c: Call_DebuggerC
xnu/osfmk/kdp/ml/ppc/kdp_machdep.c: kdp_trap
```

Back to Listing 1

Line 21 `xnu/osfmk/kdp/kdp_udp.c: kdp_raise_exception`

This line contains an exception message followed by the exception number, code, and subcode in parentheses.

`exception`: The array `kdp_trap_codes` defined in `xnu/osfmk/kdp/ml/ppc/kdp_machdep.c` is used to convert PowerPC-specific exception codes to the generic Mach exception codes used by KDB (kernel debugger). The Mach codes are defined in `mach/exception_type.h`, but this isn't generally useful because most unhandled PowerPC exceptions are mapped to `EXC_BAD_ACCESS` (1).

`exception_message`: The Mach exception code is used to look up a text message describing the exception. The message table `exception_message` is defined in `xnu/osfmk/kdp/kdp_udp.c`.

`code` and `subcode` are not used and are always zero.

Back to Listing 1

Line 22 `xnu/osfmk/kdp/kdp_udp.c: kdp_connection_wait`

This is the built-in Ethernet MAC address of the panicked machine. This and the IP address (line 23) are used to establish a remote debugging session.

Back to Listing 1

Line 23 `xnu/osfmk/kdp/kdp_udp.c: kdp_connection_wait`

This is the IP address of the panicked machine. This and the Ethernet MAC address (line 22) are used to establish a remote debugging session.

Back to Listing 1

Line 24 `xnu/osfmk/kdp/kdp_udp.c: kdp_connection_wait`

Back to Listing 1

Line 25 `xnu/osfmk/kdp/kdp_udp.c: kdp_connection_wait`

At this point the system is waiting for a connection from a remote debugger.

Back to Listing 1

Back to top

Isolating the Crash

Assume that one of your customers or testers had your kernel extension installed and experienced a kernel panic. Fortunately they sent you the panic dump shown earlier. How would you go about finding the cause of the crash?

Start by running the same version of the operating system as on the panicked machine. Use the kernel and kext version numbers from the panic dump to confirm that you're running the correct versions.

Next, take a quick glance at the kind of crash and in which kernel extension the crash occurred. In our example, a data access exception occurred with the program counter containing 0x0E692550. Looking at the list of loaded kernel extensions, the closest match is `com.apple.dts.driver.PanicDriver` which is loaded starting at address 0x0E691000. Then, because this is a data access exception, DAR contains the address which could not be accessed. In this case, it was an attempt to access memory at 0xDEADBEEF that triggered the exception.

The backtrace can be used to get a more precise picture of the sequence of calls that led up to the crash. To decipher the backtrace, it's necessary to create relocated symbol files for the kernel and each kernel extension listed in the backtrace. A new set of symbol files must be generated each time a kernel extension is loaded because the kext's load address is likely to be different each time.

Starting with Mac OS X version 10.2, generating symbol files is done via the `kextload` command as illustrated in Listing 3. The `-s` specifies the directory where to write the symbol files. The `-n` option causes `kextload` to prompt for the load address of each kernel extension and its dependencies.

Listing 3. Generating the symbol file using `kextload`

```
[localhost:~] me% sudo kextload -s /tmp -n PanicDriver/build/PanicDriver.kext/
Password:
kextload: notice: extension PanicDriver/build/PanicDriver.kext/ has debug properties set
...some output elided...
enter the hexadecimal load addresses for these modules:
com.apple.iokit.IOUSBFamily: 0xed9c000
com.apple.dts.driver.PanicDriver: 0xe696000
```

This results in a separate symbol file for each module, named `<module-name>.sym`

On versions of Mac OS X prior to 10.2, this is done via the `kmodsyms` command as illustrated in Listing 4. Note that it is necessary to provide the load address of each kernel extension and its dependencies. If there are multiple dependencies, each dependency is entered with a separate `-d` option. The `-v` option produces the verbose output also shown in Listing 4.

Listing 4. Generating the symbol file using `kmodsyms`

```
[localhost:~] me% kmodsyms -v -k /mach_kernel \
```

```
-d /System/Library/Extensions/IOUSBFamily.kext/Contents/MacOS/IOUSBFamily@0xed9c000 \
-o /tmp/com.apple.dts.driver.PanicDriver.sym
PanicDriver/build/PanicDriver.kext/Contents/MacOS/PanicDriver@0xe691000
kmodsyms: Returning fake load address of 0x ed9cb10
kmodsyms: kmod name: com.apple.iokit.IOUSBFamily
kmodsyms: kmod start @ 0xedab39c
kmodsyms: kmod stop  @ 0xedab408
kmodsyms: Returning fake load address of 0x e691b10
kmodsyms: kmod name: com.apple.dts.driver.PanicDriver
kmodsyms: kmod start @ 0xe692574
kmodsyms: kmod stop  @ 0xe6925e0
[localhost:~] me%
```

Your own kernel extensions will have full line number and function name information provided they were built using Project Builder's Development build style. Other kernel code will contain just enough name information to link against.

Next, load the symbol files into GDB using the `add-symbol-file` command as demonstrated in Listing 5.

Listing 5. Loading the symbol file into GDB

```
[localhost:~] me% gdb /mach_kernel
GNU gdb 5.1-20020408 (Apple version gdb-228) (Sun Jul 14 10:07:24 GMT 2002)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-macos10".
(gdb) add-symbol-file /tmp/com.apple.dts.driver.PanicDriver.sym
add symbol table from file "/tmp/com.apple.dts.driver.PanicDriver.sym" at
(y or n) y
Reading symbols from /tmp/com.apple.dts.driver.PanicDriver.sym...done.
(gdb)
```

If you have more than one symbol file, as would be typical with Mac OS X 10.2 or later, repeat the `add-symbol-file` command for each one.

In the case of I/O Kit C++ function names, you may find it helpful to unmangle the names to make them more readable. The command `set print asm-demangle on` is a handy way to do this. This command controls the demangling of C++ and Objective-C names in disassembly listings.

Display the instruction located at the program counter (PC) using the "examine memory" command `x/i <address>`. Depending on the type of exception, this will either be the instruction that caused the exception or the one immediately following. An example is shown in Listing 6.

Listing 6. Disassembling from the program counter.

```
(gdb) set print asm-demangle on
(gdb) x/i 0xe692550
0xe692550 <com_apple_dts_driver_PanicDriver::start(IOService*)+276>:    stw     r0,0(r9)
(gdb)
```

Next, for each address given in the backtrace, display the instruction located immediately prior to that address using the command `x/i <address>-4`. This will yield the name of the function in which the address is located. Note that each instruction disassembled from the backtrace should be some form of branch instruction. To understand why, recall that the backtrace is a listing of the return addresses saved prior to executing a function call. If the disassembly shows something other than a branch instruction, this is a clue that you may not have generated your symbol file correctly, or that the operating system version is not the same as on the panicked machine.

Listing 7 shows the results of decoding the backtrace shown in Listing 1.

Listing 7. Decoding the backtrace.

```
(gdb) x/i 0x0e6974a8-4
0xe6974a4 <com_apple_dts_driver_PanicDriver::start(IOService*)+104>:    bl
0xeb4f5b0 <com_apple_dts_driver_PanicDriver::start(IOService*)+372>

(gdb) x/i 0x0213a88-4
0x213a84 <IOService::startCandidate(IOService*)+116>:   bctrl

(gdb) x/i 0x213884-4
0x213880 <IOService::probeCandidates(OSOrderedSet*)+2096>:      bctrl

(gdb) x/i 0x2141d4-4
0x2141d0 <IOService::doServiceMatch(unsigned long)+452>:        bctrl

(gdb) x/i 0x214830-4
0x21482c <_IOConfigThread::main(_IOConfigThread*)+280>: bctrl

(gdb) x/i 0x204cb0-4
0x204cac <ioThreadStart+56>:    bctrl

(gdb) x/i 0x204c74-4
0x204c70 <IOLibInit+184>:       blr
(gdb)
```

Back to top

Summary

Using the techniques discussed in this technote, it is possible to perform an effective post-mortem analysis of a kernel panic. While the information in a panic dump may have been cryptic at first, it should now be just another debugging tool available to the Mac OS X developer.

Back to top

References

[1] *The Design and Implementation of the 4.4BSD Operating System*, McKusick et al., Addison-Wesley, 1996.

PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors, IBM Microelectronics document G522-0290-01 revised 02/21/2000.

Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture, Motorola document MPCFPE32B/AD, REV 2, revised 12/2001

Back to top

Downloadables

	Acrobat version of this Note (140K)	Download
	PanicDriver sample code (8K)	Download

Back to top
