## Technical Note TN2123

# CrashReporter

CrashReporter is a debugging facility in Mac OS X that logs information about all programs that crash. This technote describes CrashReporter in detail. It includes a description of the crash logs generated by CrashReporter, and how you can use these logs to debug your program.

This technote is useful for anyone who develops Mac OS X user space software.



## Introduction

Mac OS X's CrashReporter is a useful facility for learning about problems your application is experiencing in the field. CrashReporter performs two useful actions:

- When a program crashes, CrashReporter will record a **crash log** (typically into `~/Library/Logs/CrashReporter/`), and inform the user by logging a message to the system logging facility.
- In addition, if the program that crashed is running as a logged in GUI user, CrashReporter will present the user with a dialog asking them whether they want to submit a bug report to Apple (see Figure 1). If the user clicks the Report button, CrashReporter displays another dialog that shows the details of the report (see Figure 2) and allows them to comment it before submission.

**Figure 1:** First CrashReporter dialog

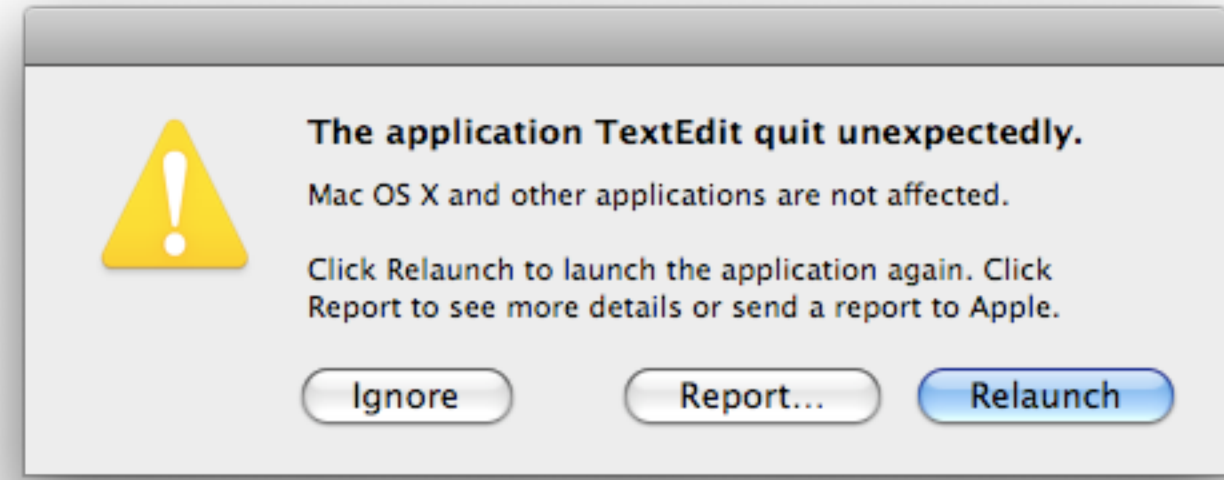
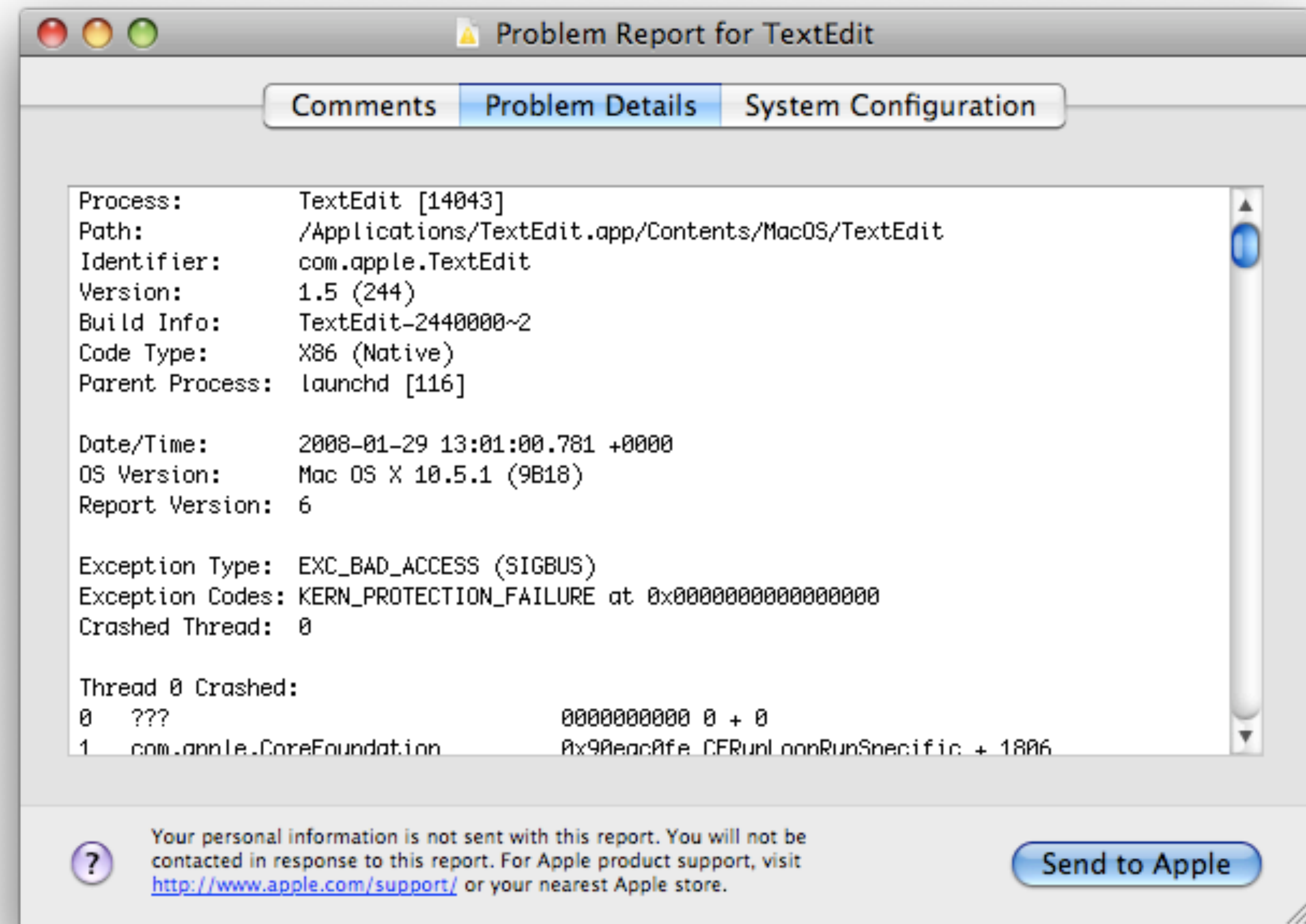



**Figure 2:** Second CrashReporter dialog




In this technote I explain how to interpret crash logs that you have obtained from end users. In the first section I explain each part of the crash log in detail. Following that I show you how you can get useful information from a crash log even if your program ships without debugging symbols. Then I explain how to use CrashReporterPrefs to customize CrashReporter's behavior. Finally, I explain some limitations of the current implementation.

**IMPORTANT:** This technote describes CrashReporter as it's implemented in Mac OS X 10.5. CrashReporter has evolved over time, and there are numerous differences between the current version and earlier ones. I've called out these changes where they are significant.

**Note:** CrashReporter has limited support for some deprecated technologies, most notably the PEF binary image format. This technote does not describe that support.

## Crash Log Placement

CrashReporter usually places the crash log in the user's home directory, as explained above. However, under some circumstances it will put the crash log in `/Library/Logs/CrashReporter/`. These include:

- if it can't determine the ownership of the crashed process
- if the crashed process was owned by root
- if the user's home directory is not available or not writable

Each crash log is written to a separate file. The file name is of the form `PPP_YYYY-MM-DD-HHMMSS_NNN.crash`, where PPP is the process name; YYYY, MM, DD, HH, MM, and SS are the date and time of the crash; and NNN is the host name. For example, `TextEdit_2008-01-29-143702_guy-smiley.crash` is a TextEdit crash from 29 Jan 2008 at 14:37:02 on the machine "guy-smiley". To prevent unbounded disk use, CrashReporter limits the number of crash log files for any given combination of user, process name and host name. The current limit is 20.

> **IMPORTANT:** CrashReporter also creates other files within the CrashReporter directory. Specifically, it creates files of the form `.PPP_NNN_CrashHistory.plist` (for example, `.TextEdit_guy-smiley_CrashHistory.plist`). These files are invisible in the Finder. You must not rely on the presence or format of these files.

> **Note:** Prior to Mac OS X 10.5 CrashReporter created files of the form `PPP.crash.log`, where PPP is the process name. All crash logs for a given process name were appended to that file. For example, all TextEdit crashes were logged in `TextEdit.crash.log`. There was nothing to prevent these files from growing without bound.

Back to Top ▲

## CrashReporter Logging

CrashReporter logs via Apple System Log. All CrashReporter log entries have the Facility set to "Crash Reporter". You can display all such message with the command shown in Listing 1.

**Listing 1:** Displaying all CrashReporter log entries

```
$ syslog -k Facility eq "Crash Reporter"
[…]
[…] Formulating crash report for process TextEdit[9803]
[…] Saved crashreport to /Users/quinn/Library/Logs/CrashReporter/\
TextEdit_2008-01-29-204411_guy-smiley.crash using uid: 2000 gid: 2000, \
euid: 2000 egid: 2000
```

You can display recent log entries and wait for more to show up (much like `tail -f`), with the command shown in Listing 2.

**Listing 2:** Waiting for new CrashReporter log entries

```
$ syslog -w -k Facility eq "Crash Reporter"
[…]
```

You can do the same thing from the Console application by choosing New Log Database Query from the File menu and then configuring the query to search for log messages where the Facility is "Crash Reporter".

> **Note:** Prior to Mac OS X 10.5 CrashReporter logged to both the system log (`/var/log/system.log`) and its own specific log file (`/var/log/crashreporter.log`).

Back to Top

## Crash Log Versions

Crash logs include a version number that you can use to interpret the information contained in the log. This version number is loosely related to the system version. Table 1 shows that relationship.

**Table 1:** Crash Log Versions

| Crash Log Version | Mac OS X Versions |
|---|---|
| 1 | prior to 10.3.2 |
| 2 | 10.3.2 through 10.3.9 |
| 3 | 10.4.x on PowerPC |
| 4 | 10.4.x on Intel |
| 5 | none |
| 6 | 10.5 and later |

Back to Top

# Anatomy of a Crash Log

A crash log has a number of different parts; in the following sections I describe each part in detail.

## Process Information

The first part of the crash log contains information about the process that crashed, as illustrated in Listing 3.

**Listing 3:** Process information

```
Process:          TextEdit [8752]
Path:             /Applications/TextEdit.app/Contents/MacOS/TextEdit
Identifier:       com.apple.TextEdit
Version:          1.5 (244)
Build Info:       TextEdit-2440000~2
Code Type:        X86 (Native)
Parent Process:   launchd [241]
```

The most important thing to note here is the name of the process that crashed. In some cases the actual process that died is not what you think. For example, if your application uses a helper tool to do some work, and that helper tool dies, you want to focus on the helper tool's code and not waste time debugging the application code.

The "Process" field includes the name and PID (in square brackets) of the crashed process. The "Path" field is the path to the process's executable.

The "Identifier" field contains the bundle identifier, if any, of the crashed process.

CrashReporter gets the "Version" field from the process's executable. If the process is a packaged application, the version is composed of the `CFBundleShortVersionString` and `CFBundleVersion` properties from its `Info.plist` file. If the process is single file application, the version is derived from its `'vers'` ID=1 resource.

The "Build Info" field shows information extracted from the `version.plist` file in the application's bundle, if any. This file, and hence this field, is typically only present for Apple applications.

The "Code Type" field shows the type of code that was executing. If your program is a universal binary, you should check this field to see which architecture was actually being run (for example, the user might have accidentally run your program using Rosetta).

The "Parent Process" field includes the name and the PID (in square brackets) of the parent process. It's worth checking that this is what you'd expect it to be.

**Note:** In crash logs prior to version 6 the "Process" field was known as the "Command" field, and the process ID was included in a separate "PID" field.

**Note:** The "Path" and "Version" fields were introduced with version 2 crash logs.

**Note:** The "Identifier" field was introduced with version 6 crash logs.

**Note:** In crash logs prior to version 6 the information in the "Build Info" field was split across three separate fields: "Build Version", "Project Name" and "Source Version". These fields were introduced with version 3 crash logs.

**Note:** The "Code Type" field was introduced with version 6 crash logs.

**Note:** In crash logs prior to version 6 the "Parent Process" field was named "Parent". It was first included in version 3 crash logs.

**Note:** In crash logs prior to version 6 the process information part of the crash log was placed after the basic information part.

Back to Top

## Basic Information

The next part of the crash log contains information about the crash log itself. You can see an example in Listing 4.

**Listing 4:** Basic information

```
Date/Time:       2008-01-29 12:32:46.239 +0000
OS Version:      Mac OS X 10.5.1 (9B18)
Report Version:  6
```

The most important piece of information here is the OS version. You should pay particular attention to the build number; each user-visible version of Mac OS X can have multiple variants distinguished only by their build numbers (this typically happens for hardware-specific system releases). In addition, make sure to look at the time and date to see if there are any suspicious patterns: if you get lots of crash logs that all occur at 12:00, you probably need to investigate your time handling code.

The "Report Version" field is discussed in Crash Log Versions.

**Note:** Version 2 crash logs included a "Host Name" field in this section; this is not present in later versions.

Back to Top

## Exception Information

The third part of the crash log shows information about the processor exception that was the immediate cause of the crash. Listing 5 shows a typical example.

**Listing 5:** Exception information

```
Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Codes: KERN_PROTECTION_FAILURE at 0x0000000000000000
Crashed Thread:  0
```

The "Crashed Thread" field denotes the thread that crashed; it is redundant because the backtrace section highlights the crashing thread.

> **Note:** The "Crashed Thread" field was introduced in version 2 crash logs. Prior to version 6 crash logs it was called "Thread" and it appeared in the process information section.

The most common forms of exception are:

- `EXC_BAD_ACCESS/KERN_INVALID_ADDRESS` — This is caused by the thread accessing unmapped memory. It may be triggered by either a data access or an instruction fetch; the Thread State section describes how to tell the difference.
- `EXC_BAD_ACCESS/KERN_PROTECTION_FAILURE` — This is caused by the thread trying to write to read-only memory. This is always caused by a data access.
- `EXC_BAD_INSTRUCTION` — This is caused by the thread executing an illegal instruction.
- `EXC_ARITHMETIC/EXC_I386_DIV` — This is caused by the thread doing an integer divide by zero on an Intel-based computer.

For **memory access exceptions** (`EXC_BAD_ACCESS`) the exception part of the crash log contains the address that triggered the exception (the **exception address**). In Listing 5 that address is 0x0000000000000000.

Back to Top

## Backtrace Information

The fourth part of the crash log, which displays a backtrace for all of the threads in the crashed process, is typically the most interesting. Listing 6 shows an example.

**Listing 6:** Backtrace information

```
Thread 0 Crashed:
0   ???                           0000000000 0 + 0
1   com.apple.CoreFoundation      0x942cf0fe CFRunLoopRunSpecific + 18…
2   com.apple.CoreFoundation      0x942cfd38 CFRunLoopRunInMode + 88
3   com.apple.HIToolbox           0x919e58a4 RunCurrentEventLoopInMode…
4   com.apple.HIToolbox           0x919e56bd ReceiveNextEventCommon + …
5   com.apple.HIToolbox           0x919e5531 BlockUntilNextEventMatchi…
6   com.apple.AppKit              0x9390bd5b _DPSNextEvent + 657
7   com.apple.AppKit              0x9390b6a0 -[NSApplication nextEvent…
8   com.apple.AppKit              0x939046d1 -[NSApplication run] + 79…
9   com.apple.AppKit              0x938d19ba NSApplicationMain + 574
10  com.apple.TextEdit            0x00001df6 0x1000 + 3574
```

In this example there is only one thread, so there's only one backtrace. In a multi-threaded process, there is one backtrace per thread. Thus, it's critical that you identify the thread that crashed. CrashReporter makes this easy by tagging that backtrace with the text "Thread <ThreadNumber> Crashed:". However, it's easy to overlook this text and erroneously assume that the Thread 0 is the one that crashed.

**Note:** Your process may be multi-threaded even if you don't explicitly create any threads. Various frameworks can create threads on your behalf. For example, CFSocket creates a thread to integrate sockets with the runloop.

Each line of the backtrace describes a nested function invocation (a **frame**), with the most recently executed function at the top and the least recently executed at the bottom. For each frame, the columns in the backtrace are as follows.

- The first column is the frame number, starting at 0 (indicating the function that crashed) and incrementing for each nested function call.
- The second column is the name of the binary image containing the code executing in this frame; this is derived by cross referencing the program counter address (from the next column) with the list of loaded binary images.
- The third column is the program counter address within the frame. For frame 0 this is typically the address of the instruction that caused the exception. For higher frames this is the return address for that frame. That is, for frame N it points to the next instruction that will execute when the function referenced by frame N - 1 returns.
- The fourth column is the symbolic name for the program counter address given in the third column. If you strip debugging symbols before shipping your application to end users, this column will just contain a hex number. You can work out the corresponding symbolic name using the technique described later in this document.

Finally, if your program is multi-threaded, you can often identify which thread is which by looking at the symbolic names deep within the backtrace. For example, in Listing 6, frame 9 lists `NSApplicationMain` as its symbolic address, indicating that this thread is the main thread. In contrast, the deepest frame for a pthread is always the routine `_pthread_start`.

Back to Top

## Thread State

The next part of the crash log contains a dump of the processor state of the thread that crashed. Listing 7 shows an example of this for PowerPC.

**Listing 7:** PowerPC thread state

```
Thread 0 crashed with PPC Thread State 64:
  srr0: 0x0000000000000000 srr1: 0x000000004000d030                    …
    cr: 0x44022282              …                  lr: 0x000000009000a6bc…
    r0: 0x00000000ffffffe1   r1: 0x00000000bfffeb10   r2: 0x00000000a…
    r4: 0x0000000003000006   r5: 0x0000000000000000   r6: 0x000000000…
    r8: 0x0000000000000000   r9: 0x0000000000000000  r10: 0x000000000…
   r12: 0x000000009000a770  r13: 0x0000000000000000  r14: 0x000000000…
   r16: 0x0000000000000000  r17: 0x0000000000000000  r18: 0x000000000…
   r20: 0x00000000101a7026  r21: 0x00000000be5b19d8  r22: 0x000000000…
   r24: 0x0000000000000450  r25: 0x0000000000001203  r26: 0x000000000…
   r28: 0x0000000000000000  r29: 0x0000000003000006  r30: 0x000000000…
```

To get the most out of this information, you need a good understanding of the Mac OS X application binary interface (ABI) for the processor. For a detailed description, see Mac OS X ABI Function Call Guide. However, there are some simple ways get useful results without a full understanding of the ABI.

## PowerPC Architecture

For PowerPC-based computers, you should consider the following points:

- Focus on three values: `srr0`, `lr`, and the exception address (described earlier).
- `srr0` is the program counter at the time that the exception occurred. That is, it's the address of the instruction that caused the exception. For most non-memory access exceptions (for example, `EXC_BAD_INSTRUCTION` caused by executing an illegal instruction), this is the key value. For memory access exceptions, it's only part of the equation.
- `lr` is typically used to hold the return address of a function call.
- For memory access exceptions:
  - If `srr0` is equal to the exception address, the exception was caused by fetching instructions. Typically this means that you've called a bogus function pointer (or, equivalently, called a method on a bogus object). In this case the return address is typically in `lr`, which tells you the address of the code that called the bogus function pointer.
  - Furthermore, if `srr0` is equal to `lr` which is equal to the exception address, your program has crashed returning from a function. This typically means that you've corrupted the stack (the return address is typically saved on the stack during the execution of a function) and then died returning to a bogus address.
  - If `srr0` is not equal to the exception address, the exception was caused by a memory access instruction (in terms of C, this means that you're dereferencing an invalid pointer).
- The function call ABI for PowerPC code is registered based; thus, if you crash early in the execution of a function, you may be able to see its parameters in the registers. See Mac OS X ABI Function Call Guide for details.
- Finally, it can be helpful to look through the other registers for telltale signs. For example, if a register contains ASCII characters whose value only appear in one place in your program, that's a clue as to what code has executed recently. Alternatively, if a register contains a well known error value (for example, `dskFulErr`, which means "disk full error" in the Core Services File Manager, whose value is -34, or 0xffffffde in hex), that might be a clue as to why your program failed.

In the example in Listing 7 (which is the thread state for a memory access exception), you can see that `srr0` is 0x00000000, which is equal to the exception address (see Exception Information) but is not equal to `lr`. Thus, the program has crashed by calling a `NULL` function pointer and the caller's address is in `lr`.

> **Note:** Prior to version 3, crash logs only include the bottom 32-bits of each PowerPC register.

Back to Top

## 32-bit Intel Architecture

Listing 8 shows the thread state for an Intel-based computer running 32-bit code.

**Listing 8:** 32-bit Intel thread state

```
Thread 0 crashed with X86 Thread State (32-bit):
  eax: 0x00000000  ebx: 0x942cea07  ecx: 0xbfffed1c  edx: 0x94b3a8e6
  edi: 0x00000000  esi: 0x00000000  ebp: 0xbfffed58  esp: 0xbfffed1c
   ss: 0x0000001f  efl: 0x00010206  eip: 0x00000000   cs: 0x00000017
   ds: 0x0000001f   es: 0x0000001f   fs: 0x00000000   gs: 0x00000037
  cr2: 0x00000000
```

For Intel-based computers running 32-bit code, you should consider the following points:

- Focus on two values: `eip` and the exception address (described earlier).
- `eip` is the program counter at the time that the exception occurred. That is, it's the address of the instruction that caused the exception. For most non-memory access exceptions (for example, `EXC_ARITHMETIC/EXC_I386_DIV` caused by an integer division by zero), this is the key value.
- For memory access exceptions:

- If `eip` is equal to the exception address, the exception was caused by fetching instructions. Typically this means:
  - you've called a bogus function pointer (or, equivalently, called a method on a bogus object)
  - you've returned to a bad address which, in turn, means that you've corrupted the stack
- If `eip` is not equal to the exception address, the exception was caused by a memory access instruction (in terms of C, this means that you're dereferencing an invalid pointer).

- Finally, as with PowerPC, it can be helpful to look through the other registers for telltale signs.

> **Note:** Because of the way the Intel architecture works, it's harder to get useful results from the thread state than it is on PowerPC. For example, on the PowerPC architecture the return address is stored in a register (`lr`), while on Intel it's stored on the stack.

> **Note:** The 32-bit Intel thread state in crash logs prior to version 6 did not include the `cr2` register.

Back to Top

## 64-bit Intel Architecture

Listing 9 shows the thread state for an Intel-based computer running 64-bit code.

**Listing 9:** 64-bit Intel thread state

```
Thread 0 crashed with X86 Thread State (64-bit):
  rax: 0x0000000000000000  rbx: 0x0000000000000000  rcx: 0x00007fff5fbfec48…
  rdi: 0x00007fff5fbfed40  rsi: 0x0000000003000006  rbp: 0x00007fff5fbfeca0…
   r8: 0x0000000000001003   r9: 0x0000000000000000  r10: 0x0000000000000450…
  r12: 0x0000000000001003  r13: 0x0000000000000450  r14: 0x00007fff5fbfed40…
  rip: 0x0000000000000000  rfl: 0x0000000000010206  cr2: 0x0000000000000000
```

In general you should interpret this thread state in much the same way as you would a 32-bit Intel thread state. The key differences are:

- The PC address is in `rip`, not `eip`.
- As with PowerPC, the function call ABI for 64-bit Intel code is registered based; thus, if you crash early in the execution of a function, you may be able to see its parameters in the registers. See Mac OS X ABI Function Call Guide for details.

Back to Top

## Binary Images

The next part of the crash log is a description of all of the binary images loaded into the process. Listing 10 is an example of this.

**Listing 10:** Binary Images

```
Binary Images:
    0x1000 -     0x18feb  com.apple.TextEdit 1.5 (244) <e1480af78e2746195aa…
 0xc648000 -   0xc72eff7  com.apple.RawCamera.bundle 2.0 (2.0) /System/Libr…
0x8fe00000 - 0x8fe2d883  dyld 95.3 (???) <81592e798780564b5d46b988f7ee1a6a…
0x90046000 - 0x9004efff  com.apple.DiskArbitration 2.2 (2.2) <1551b2af557f…
0x9004f000 - 0x9004fff8  com.apple.ApplicationServices 34 (34) <8f910fa65f…
```

```
0x90056000 - 0x900affff  libGLU.dylib ??? (???) /System/Library/Frameworks…
0x900b0000 - 0x900b0ffc  com.apple.audio.units.AudioUnit 1.5 (1.5) /System…
0x900b1000 - 0x90163ffb  libcrypto.0.9.7.dylib ??? (???) <330b0e48e67faffc…
[…]
```

This list is particularly useful because you can use it to determine a symbolic backtrace in a program without symbols. It can also be useful if your program makes extensive use of plug-ins because it will show you exactly what plug-ins were loaded in your process. Finally, you can look through this list for libraries that you don't expect to be loaded into your process, such as those used by common application patching (or 'enhancement') technologies.

**Note:** This section was introduced with version 2 crash logs.

**IMPORTANT:** The value in angle brackets is the UUID of the image, if present. Image UUIDs are important when looking at crash logs without symbols.

**Note:** Binary image UUIDs were first included in version 6 crash logs.

Back to Top

## Rosetta Extras

If the crashed process was run using Rosetta, extra information is added to the crash log. To start with, the "Code Type" field in the Process Information part indicates that Rosetta was in use; an example of this is shown in Listing 11.

**Listing 11:** Process information with Rosetta

```
Process:         TextEdit [9031]
Path:            /Applications/TextEdit.app/Contents/MacOS/TextEdit
Identifier:      com.apple.TextEdit
Version:         1.5 (244)
Code Type:       PPC (Translated)
Parent Process:  launchd [241]
```

In addition, a "Translated Code Information" part is added after the Binary Images section. This section includes:

- the Rosetta version number
- the process's command line arguments
- a limited form of exception information
- for each thread, a backtrace and the PowerPC thread state, albeit in a slightly different format

Listing 12 shows an example of this information.

**Listing 12:** Translated code information

```
Translated Code Information:
Rosetta Version:  20.44
Args:    /Applications/TextEdit.app/Contents/MacOS/TextEdit -psn_0_2761378
Exception: EXC_BAD_ACCESS (0x0001)

Thread 0: Crashed (0xb7fff9d0, 0xb80bc8c8)
```

```
0x40400000: No symbol
0x90a9b35c: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a9b290: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a9a7a8: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a9a0e0: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a99a1c: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a98458: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x90a6b8f4: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x909d8ed8: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x909a9930: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit…
0x00001e18: /Applications/TextEdit.app/Contents/MacOS/TextEdit : start + …
0x00000000: /Applications/TextEdit.app/Contents/MacOS/TextEdit :   + 0

PPC Thread State
srr0: 0x00000000  srr1: 0x00000000                vrsave: 0x00000000
cr:  0xXXXXXXXX    xer: 0x00000000     lr: 0x90a9b35c    ctr: 0x0000e814
r00: 0x90a9b35c   r01: 0xbfffe5d0   r02: 0xa0bcf924   r03: 0x00234840
r04: 0x00020000   r05: 0x002adce0   r06: 0x002adce0   r07: 0x002adce0
r08: 0xa1b1c1d3   r09: 0x00000000   r10: 0x00000004   r11: 0x00000001
r12: 0x0000e814   r13: 0xa01da174   r14: 0xa01da174   r15: 0xa01da174
r16: 0xa01da174   r17: 0xa01da174   r18: 0x00000000   r19: 0x002c00b0
r20: 0xbfffe760   r21: 0xa01da174   r22: 0xa01da174   r23: 0xa01da174
r24: 0xa01ea174   r25: 0x002adce0   r26: 0x002475c0   r27: 0x00015dd4
r28: 0x00234840   r29: 0x00020000   r30: 0x00234840   r31: 0x90a9b2f0
```

Back to Top

## Crash Logs Without Symbols

Symbols have always presented a conundrum for developers:

- You want to remove all symbols from your program because this makes the code smaller, and potentially faster.
- However, if you remove all symbols, your crash log backtraces will contain just hex values, which makes it significantly harder to interpret them.

Modern versions of Xcode make it easy to have your cake and eat it to. You can remove all symbols from your program before shipping it to end users, and still have easy access to symbolic debugger information when interpreting crash logs. This section explains how to do this.

### Symbols Divided

When you build a program with Xcode, you must deal with two very different types of symbols:

- **Mach-O Symbols** — These symbols may be interpreted by the linker (either static, ld, or dynamic, dyld).
- **Debugger Symbols** — These symbols are interpreted by the debugger.

Mach-O symbols are subdivided into two groups:

- **local symbols** — These symbols are ignored by the linker, and are solely present for the benefit of tools like CrashReporter. When you declare a function as `static` in C, it gets recorded as a local symbol.
- **global symbols** — These symbols are interpreted by both the static and dynamic linker. When you declare a function as `extern` in C, it gets recorded as a global symbol.

Xcode supports two forms of debugger symbols:

- DWARF — This is the modern debugger symbols format, supported by Xcode 2.3 and later. DWARF symbols have numerous advantages, not the least being the nice integration with CrashReporter, as described below.
- STABS — This older debugger symbols format is now deprecated.

> **Note:** STABS debugger symbols are actually stored in the Mach-O symbol table. This made the process of managing symbols more confusing because it conflated two very different concepts (that is, Mach-O symbols and debugger symbols).

Back to Top

## Stripping Debugger Symbols

Xcode's DWARF implementation makes it very easy to strip debugger symbols from your released program. All you need to do is set the "Debug Information Format" build setting for your release build to "DWARF with dSYM File" (`DEBUG_INFORMATION_FORMAT = dwarf-with-dsym`). When you build your program Xcode will extract all of the debugging information and place it in a `.dSYM` document, which it places right next to your program in the build folder. You can then:

- package up your program and ship it to your users
- archive the `.dSYM` file

If, at any time in the future, you need to debug your released program, you can just put it and the `.dSYM` file in the same directory and GDB will automatically find and use the debugger symbols.

Back to Top

## Stripping Mach-O Symbols

Stripping Mach-O symbols is a more complicated issue. The problem is that some Mach-O symbols are interpreted at runtime by the dynamic linker, and you must ensure that those symbols are not stripped. There's one overall approach, but you have modify it depending on the type of program you are building.

The first step is to enable the "Deployment Postprocessing" build setting (`DEPLOYMENT_POSTPROCESSING = YES`) for your release build. This will cause Xcode to automatically remove some symbols from your program. The exact behavior depends on the "Strip Style" build setting. You have three choices for this setting:

- All Symbols (`STRIP_STYLE = all`) — Xcode will strip all symbols that are not specifically marked as being needed at runtime. This is equivalent to running strip tool with the "-u" and "-r" flags.
- Non-Global Symbols (`STRIP_STYLE = non-global`) — Xcode will strip all local symbols. This is equivalent to running strip tool with the "-x" flags.
- Debugging Symbols (`STRIP_STYLE = debugging`) — Xcode will strip no symbols! Actually, it runs the strip tool with the "-S" flag, which causes it to remove all debugger symbols. This is relevant if you're using STABS debugger symbols. However, if you're sensible and are using DWARF, your debugger symbols will not be affected by this; `strip` will remove some information from your program (notably the debug map used by dsymutil) but this will not affect the `.dSYM` file that Xcode has already created.

Critically, the default value for the "Strip Style" build setting depends on your target type. Table 2 shows this relationship.

**Table 2:** Default strip style by target type

| Target Type | Mach-O Type | Default Strip Style |
|---|---|---|
| Application, Command-Line Tool | MH_EXECUTE | All Symbols |
| Bundle | MH_BUNDLE | Non-Global Symbols |
| Framework, Dynamic Library | MH_DYLIB | Debugging Symbols |
| Static Library | MH_OBJECT | Debugging Symbols |

> **IMPORTANT:** If you do not set the "Strip Style" build setting at the project layer, Xcode will display a value of All Symbols. This is misleading. The actual value you get depends on the target type, as shown above. If you open the target build settings, you will set the actual value that will be used.
> On the other hand, if you do set a value for the "Strip Style" build setting at the project layer, this will apply to all targets (unless you override it at the target layer). If your project has multiple targets of different types, this is unlikely to be useful. In this case you will probably want to set the "Strip Style" build setting at the target layer for all targets.
> For more information about how Xcode's build settings are layered, see the Build Setting Evaluation section section of the Xcode User Guide.

There are two common approaches to stripping Mach-O symbols:

- **typical** — In most cases it's reasonable to leave all Mach-O global and local symbols in your release program. They don't bloat the program too much, and their presence can be useful. For example:
  - your CrashReporter logs will include basic symbols in the backtrace
  - you can set DTrace probe points on these symbols
- **restricted** — In some cases it's necessary to remove all non-essential symbols from a program. For example:
  - when you want to reduce the program's size as much as possible
  - when you don't want users seeing your symbol names

Implementing the typical approach is trivial. Set the "Strip Style" build setting to Debugging Symbols, and Xcode will leave all of your global and local Mach-O symbols in the program. This approach works for all target types.

Implementing the restricted approach is a bit trickier. You have to do different things depending on your target type. For an application or command-line tool, you can simply set the "Strip Style" build setting to "All Symbols".

> **Note:** This will not remove all symbols from your program. Some symbols, like `_NXArgc`, are exported as part of the C runtime system and your program will not work correctly if you strip them.

For a bundle, framework, or dynamic library, you should strip all local symbols by setting the "Strip Style" build setting to "Non-Global Symbols". You cannot, in general, strip global symbols because they may be referenced dynamically.

> **IMPORTANT:** It is possible to restrict the list of global symbols by passing various arguments to strip. However, in most cases it's easier to do this at the link stage. For example, you can explicitly control the list of exported symbols by setting the "Exported Symbols File" build setting (`EXPORTED_SYMBOLS_FILE`). Alternatively, you can use the visibility attribute to explicitly declare the visibility of symbols in your source code.

Back to Top

## Symbols and CrashReporter

Once you've set up your build system correctly, it's actually pretty easy to get meaningful results from a crash log. If you don't strip local Mach-O symbols, the backtraces in your crash log will already be decorated with the correct function names. In many cases that's all you need.

However, there are situations where you need more information.

- If you eliminate all non-essential Mach-O symbols from your program (the restricted approach, described in the previous section), the backtrace will not be decorated at all.
- Because CrashReporter does not have access to your debugger symbols, it knows nothing about your source code. Thus, when it decorates a backtrace, it can only include an offset, in bytes, from the beginning of the function. In many cases it can be hard to map that byte offset to your a line in your source code.

Fortunately, it's easy to use your `.dSYM` file to get source-level information from a crash log backtrace. The process is slightly different for position-dependent and position-independent code.

## Position-Dependent Code

For position dependent code (applications and command-line tools) the process is trivial: put your release build and your `.dSYM` file in the same directory and use GDB to get symbolic information for numeric addresses. Listing 13 shows an example of this.

**Listing 13:** Using GDB for position-dependent code

```
$ # Get the numeric values from the backtrace...
$ grep "Thread 0 Crashed:" -A 19 NoSymbolsTest_[…]_guy-smiley.crash
Thread 0 Crashed:
0   ...le.dts.NoSymbolsTest.Bundle  0x107cbf99 0x107cb000 + 3993
1   ...le.dts.NoSymbolsTest.Bundle  0x107cbfcb 0x107cb000 + 4043
2   ...dts.NoSymbolsTest.Framework  0x10005f2e 0x10005000 + 3886
3   ...dts.NoSymbolsTest.Framework  0x10005f59 0x10005000 + 3929
4   com.apple.dts.NoSymbolsTest     0x10000edf 0x10000000 + 3807
5   com.apple.AppKit                0x939dcf94 -[NSApplication sendAction…
6   com.apple.AppKit                0x939dced4 -[NSControl sendAction:to:…
7   com.apple.AppKit                0x939dcd5a -[NSCell _sendActionFrom:]…
8   com.apple.AppKit                0x939dc3bb -[NSCell trackMouse:inRect…
9   com.apple.AppKit                0x939dbc12 -[NSButtonCell trackMouse:…
10  com.apple.AppKit                0x939db4cc -[NSControl mouseDown:] + …
11  com.apple.AppKit                0x939d9d9b -[NSWindow sendEvent:] + 5…
12  com.apple.AppKit                0x939a6a2c -[NSApplication sendEvent:…
13  com.apple.AppKit                0x93904705 -[NSApplication run] + 847…
14  com.apple.AppKit                0x938d19ba NSApplicationMain + 574
15  com.apple.dts.NoSymbolsTest     0x10000e36 0x10000000 + 3638
16  com.apple.dts.NoSymbolsTest     0x10000e02 0x10000000 + 3586
17  com.apple.dts.NoSymbolsTest     0x10000d29 0x10000000 + 3369

$ # Run GDB to get the symbolic information for the address in frame 4.
$ gdb NoSymbolsTest.app
GNU gdb 6.3.50-20050815 (Apple version gdb-768) […]
(gdb) info line *0x10000edf
Line 86 of "/Users/quinn/Crash Reporter/NoSymbolsTest/AppDelegate.m" \
starts at address
0x10000edf <-[AppDelegate testAction:]+104> and ends at \
0x10000ee1 <-[AppDelegate testAction:]+106>.
```

There are a number of things to consider when doing this mapping.

- In Listing 13 I used GDB to map from address to symbol. Another option is to use atos. However, `atos` will not work correctly if you've stripped all non-essential Mach-O symbols from a program (r. 4851020).
- For the technique in Listing 13 to work correctly, you must have access to the program that the user is running, and to the matching `.dSYM` file. All three must match exactly. You can confirm this match using the program's UUID. Listing 14 shows how to get the UUID from the binary images part of the crash log, from the program itself (using dwarfdump), and from the `.dSYM` file (also using `dwarfdump`).
- The examples in this text assume that you're using the same runtime architecture as the user of your program. If that's not the case, you can use a command line option to force the tools to use the appropriate architecture. The option is typically `-arch xxx`, where `xxx` is the desired architecture (for example,

i386 or ppc), although for dwarfdump it is --arch xxx.

**Listing 14:** Using the UUID to confirm that your symbols match the user's program

```
$ # Get the UUID from the binary images part of the crash log.
$ # The UUID is displayed in angle brackets.
$ grep "0x.*com.apple.dts.NoSymbolsTest .*<" NoSymbolsTest_[…]_guy-smiley.crash
0x10000000 - 0x10000ffe  com.apple.dts.NoSymbolsTest ??? (1.0) \
<6264534bd26d5d39f7960cea770c4ea8> /Users/quinn/Crash Reporter/NoSymbolsTest/\
build/Release/NoSymbolsTest.app/Contents/MacOS/NoSymbolsTest
$ # Get the UUID from the binary that we have.
$ dwarfdump --uuid NoSymbolsTest.app/Contents/MacOS/NoSymbolsTest
UUID: 6264534B-D26D-5D39-F796-0CEA770C4EA8 (i386) NoSymbolsTest.app[…]
UUID: AA201B24-D09B-49E2-55E5-AB15AF63B12A (ppc) NoSymbolsTest.app[…]
$ # Get the UUIDs from the .dSYM file.
$ dwarfdump --uuid NoSymbolsTest.app.dSYM
UUID: 6264534B-D26D-5D39-F796-0CEA770C4EA8 (i386) NoSymbolsTest.app.dSYM
UUID: AA201B24-D09B-49E2-55E5-AB15AF63B12A (ppc) NoSymbolsTest.app.dSYM
$ # Note that all three UUIDs match!
```

Back to Top

## Position-Independent Code

Things are a little more complication for position-independent code, like frameworks, dynamic libraries or bundles. In this case you need to work out the difference between where the code was meant to be loaded and where the code was actually loaded. This value is known as the **slide**.

To find the address that the program was actually loaded, look in the binary images part of the crash log. In the example in Listing 15, the program has three relevant binary images (the main program, a framework, and a bundle) and, for each image, the first column of the output is the address where the __TEXT segment was loaded.

**Listing 15:** Getting the actual load address

```
$ # Determine the addresses that the programs were loaded.
$ grep "0x.*com.apple.dts" NoSymbolsTest_[…]_guy-smiley.crash
0x10000000 - 0x10000ffe  com.apple.dts.NoSymbolsTest ??? (1.0) […]
0x10005000 - 0x10005ffd  com.apple.dts.NoSymbolsTest.Framework ??? (1.0) […]
0x107cb000 - 0x107cbffc  com.apple.dts.NoSymbolsTest.Bundle ??? (1.0) […]
```

You can then use otool to get the intended load address of the __TEXT segment. Listing 16 shows an example of this.

**Listing 16:** Getting the intended load address

```
$ otool -l NoSymbolsTest.app/Contents/MacOS/NoSymbolsTest \
| grep -B 3 -A 2 -m 1 "__TEXT"
Load command 1
      cmd LC_SEGMENT
  cmdsize 192
  segname __TEXT
   vmaddr 0x10000000
```

```
   vmsize 0x00001000
$ otool -l NoSymbolsTest.app/Contents/Frameworks/Framework.framework/Framework \
| grep -B 3 -A 8 -m 1 "__TEXT"
Load command 0
      cmd LC_SEGMENT
  cmdsize 192
  segname __TEXT
   vmaddr 0x01000000
   vmsize 0x00001000
$ otool -l NoSymbolsTest.app/Contents/Resources/Bundle.bundle/Contents/MacOS/Bundle \
| grep -B 3 -A 8 -m 1 "__TEXT"
Load command 0
      cmd LC_SEGMENT
  cmdsize 192
  segname __TEXT
   vmaddr 0x00000000
   vmsize 0x00001000
```

Calculating the slide is now a matter of basic arithmetic.

**Table 3:** Calculating The Slide

| Program | Actual Load Address (A) | Intended Load Address (I) | Slide (A - I) |
|---|---|---|---|
| main executable | 0x10000000 | 0x10000000 | 0 |
| framework | 0x10005000 | 0x01000000 | 0x0F005000 |
| bundle | 0x107cb000 | 0x00000000 | 0x107cb000 |

**IMPORTANT:** The main executable is almost always position-dependent and thus its slide is zero. This is what makes the technique described in Position-Dependent Code so much simpler.

If you're mapping symbols using `atos`, you can feed this slide into the program using the `-s` option. If you're using GDB, the situation is more complicated. Listing 17 shows one technique for doing this.

**Listing 17:** Using GDB for position-independent code

```
$ # Get the addresses from the backtrace.
$ grep "Thread 0 Crashed:" -A 5 NoSymbolsTest_2008-02-04-111412_guy-smiley.crash
Thread 0 Crashed:
0   ...le.dts.NoSymbolsTest.Bundle  0x107cbf99 0x107cb000 + 3993
1   ...le.dts.NoSymbolsTest.Bundle  0x107cbfcb 0x107cb000 + 4043
2   ...dts.NoSymbolsTest.Framework  0x10005f2e 0x10005000 + 3886
3   ...dts.NoSymbolsTest.Framework  0x10005f59 0x10005000 + 3929
4   com.apple.dts.NoSymbolsTest     0x10000edf 0x10000000 + 3807
$ # Now map the addresses for the frames in the bundle (0 and 1).
$ # Run GDB with no arguments.
$ gdb
GNU gdb 6.3.50-20050815 (Apple version gdb-768) […]
(gdb) # Disable shared library preloading.  See below for why.
```

```
(gdb) set sharedlibrary preload-libraries off
(gdb) # Target the bundle.
(gdb) file Bundle.bundle/Contents/MacOS/Bundle
Reading symbols from […]
(gdb) # Subtract the bundle slide from the frame 0 address and then map it.
(gdb) p/x 0x107cbf99-0x107cb000
$1 = 0xf99
(gdb) info line *$1
Line 28 of "/Users/quinn/Crash Reporter/NoSymbolsTest/Bundle.m" starts at address \
0xf94 <-[Bundle testInner]+50> and ends at \
0xfa5 <-[Bundle testInner]+67>.
(gdb) # Subtract the bundle slide from the frame 1 address and then map it.
(gdb) p/x 0x107cbfcb-0x107cb000
$2 = 0xfcb
(gdb) info line *$2
Line 34 of "/Users/quinn/Crash Reporter/NoSymbolsTest/Bundle.m" starts at address \
0xfcb <-[Bundle testOuter]+36> and ends at \
0xfd1 <-[Bundle testOuter]+42>.
(gdb) quit
$ # Now do the same for the framework.
$ gdb
GNU gdb 6.3.50-20050815 (Apple version gdb-768) […]
(gdb) # Disable shared library preloading.
(gdb) set sharedlibrary preload-libraries off
(gdb) # Target the framework.
(gdb) file Framework.framework/Framework
Reading symbols from […]
(gdb) # Subtract the framework slide from the frame 2 address and then map it.
(gdb) p/x 0x10005f2e-0x0F005000
$1 = 0x1000f2e
(gdb) info line *$1
Line 39 of "/Users/quinn/Crash Reporter/NoSymbolsTest/Framework.m" starts at address \
0x1000f2e <-[Framework testInner]+308> and ends at
0x1000f35 <-[Framework testOuter]>.
(gdb) # Subtract the framework slide from the frame 3 address and then map it.
(gdb) p/x 0x10005f59-0x0F005000
$2 = 0x1000f59
(gdb) info line *$2
Line 44 of "/Users/Crash Reporter/NoSymbolsTest/Framework.m" starts at address \
0x1000f59 <-[Framework testOuter]+36> and ends at \
0x1000f5f <-[Framework testOuter]+42>.
```

**IMPORTANT:** By default, when you target a file in GDB (using the `file` command), GDB will load the symbols for that file and for all of the shared libraries that it references. This can cause problems in situations like this, where the shared libraries might overlap the program whose symbols you care about. The `set sharedlibrary preload-libraries off` command prevents GDB from loading symbols from shared libraries. This has the added benefit of making things go faster.

**Note:** The program used in the examples in this section was carefully constructed to exercise some interesting edge cases. However, that produces some effects that are quite atypical. Most notably, when building a framework you typically set its intended load address to zero, or a value that does not overlap your main executable. Thus, a framework typically has either a zero intended load address or a zero slide.

**Note:** This section focuses on the `__TEXT` segment because the symbols in a backtrace typically come from that segment. For most Mach-O images the `__TEXT` and `__DATA` segments slide together; once you work out the slide for the `__TEXT` segment, you can apply the same slide to map symbols in the `__DATA` segment. However, in Mac OS X 10.5 and later it's possible for these segments to slide independently. This typically only happens for commonly-used system frameworks.
The bad news is that the crash log does not contain the base address of the `__DATA` segment, so it's impossible to calculate the data slide of an image purely from the crash log (r. 5734989). The good news is that this is rarely necessary because:

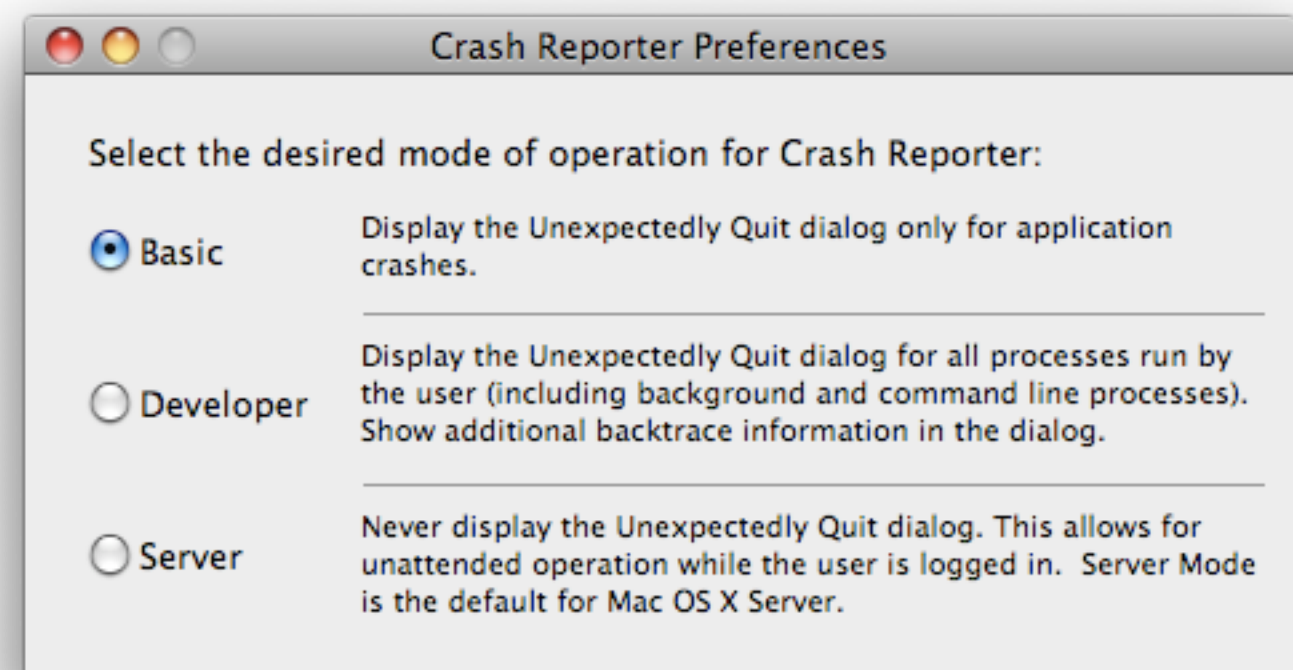
- this issue only affects frameworks that are placed in the dyld shared region (that is, commonly-used system frameworks)
- it is rare for data segment addresses to appear in a backtrace
- if they do, and the address is in a system framework, CrashReporter will have already mapped the address to a symbol

Back to Top

## CrashReporterPrefs

The CrashReporterPrefs application (installed as part of the Xcode developer tools) allows you to control how CrashReporter operates. Figure 3 shows its primary user interface.

**Figure 3:** CrashReporterPrefs user interface

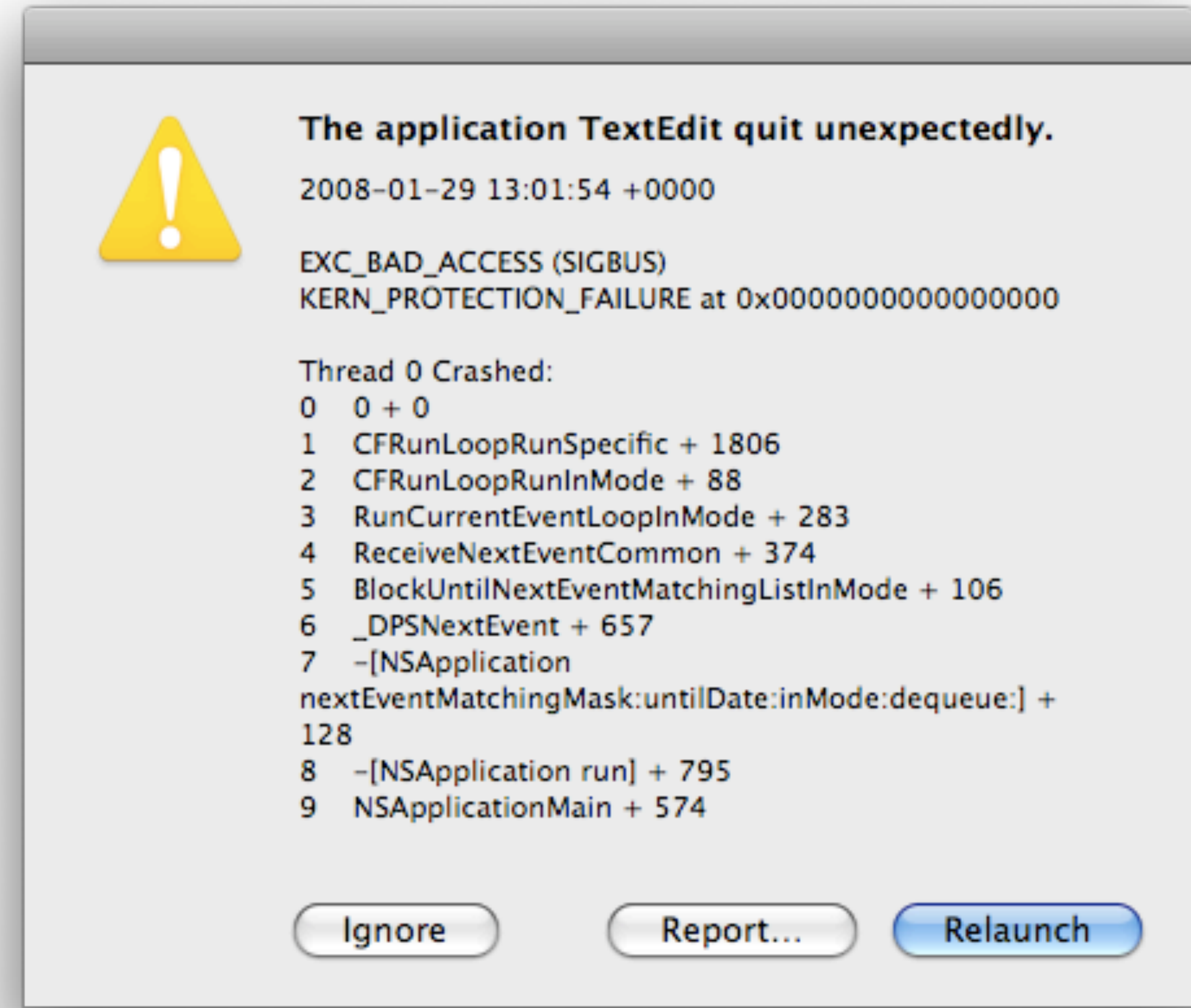



There are three modes:

- Basic — This is the default mode, described earlier.
- Developer — This mode is designed for software developers. In this mode CrashReporter will display a more detailed crash report dialog in more cases. Figure 4 shows an example of the first CrashReporter dialog when running in Developer mode.

- Server — This mode is designed for unattended servers. The CrashReporter user interface is never shown, although crash logs are still written to disk.

**Figure 4:** CrashReporter in Developer mode




**Note:** CrashReporterPrefs was introduced with Mac OS X 10.4 (Xcode 2.0). Prior to Mac OS X 10.4, there was no equivalent to Developer mode. You could, however, modify some CrashReporter behavior using hidden preferences, as described in Technical Q&A QA1288, 'Suppressing the "unexpectedly quit" alert'.
In Mac OS X 10.4.x Developer mode would offer you the option of attaching to the crashed process using GDB. This feature was removed because architectural changes in the system, designed to make crash reporting more reliable in general, made it very hard to implement.

Back to Top

## CrashReporter Limitations

CrashReporter currently has a number of limitations.

- There is currently no way for third party developers to access the reports submitted via CrashReporter. Apple is aware that there is strong demand for such a facility (r. 3356232). In fact, various third party developers have implemented their own crash reporting mechanisms: these range from the simple (have the application look at its own crash log file at launch time; if it has changed, offer to submit it to the developer) to the exceeding complex (completely reimplement CrashReporter).
- Adding some stack information to the crash log would make it more useful when debugging certain types of crashes (r. 3310695).

A number of issues that affected previous versions of CrashReporter were were addressed in Mac OS X 10.5.

- Prior to Mac OS X 10.5, CrashReporter did not generate a crash log if your program terminated because of an `abort` system call (r. 3291139).

- Prior to Mac OS X 10.5, if you wrote a program that cause an exception but handled that exception via a signal handler, CrashReporter would erroneously generate a crash log for your program (r. 2941263).

Back to Top

## Further Reading

- Technical Q&A QA1288, 'Suppressing the "unexpectedly quit" alert'
- Mac OS X ABI Function Call Guide
- Xcode User Guide
- Controlling Symbol Visibility
- Debugging with GDB
- ReportCrash man page
- Apple System Log (ASL) man page
- atos man page
- strip man page
- otool man page
- dwarfdump man page

Back to Top

## Downloadables

- Sample crash logs ("tn2123_SampleCrashLogs.zip", 26.2K)

Back to Top

## Document Revision History

| Date | Notes |
|---|---|
| 2008-04-01 | Corrected a long-standing mistyping of the srr0 register name. |
| 2008-02-27 | Updated for Mac OS X 10.5, including a description of version 6 crash logs and a complete rewrite of the "Crash Logs Without Symbols" section to account for DWARF. |
| 2006-02-28 | Updated for Mac OS X 10.4.4 on both PowerPC- and Intel-based computers. Included a description of version 3 and version 4 crash logs, and CrashReporterPrefs. |
| 2004-09-09 | Describes CrashReporter and how to debug with crash logs. |