

Technical Note TN2151

Understanding and Analyzing iOS Application Crash Reports

When an application crashes, a "crash report" is created which is very useful for understanding what caused the crash. This document contains essential information about how to symbolicate, understand, and interpret crash reports.

- Introduction
- Understanding Low Memory Reports
- Analyzing Crash Reports
 - Symbolication
 - Exception codes
- Related Documents
- Document Revision History

Introduction

When an application crashes on an iOS device, a "crash report" is created and stored on the device. Crash reports describe the conditions under which the application terminated, in most cases including a complete stack trace for each executing thread, and are typically very useful for debugging issues in the application. If you are an iOS developer, you should look at these crash reports to understand what crashes your application is having, and then try to fix them.

Low memory reports differ from other crash reports in that there are no stack traces in this type of reports. When a low memory crash happens, you must investigate your memory usage patterns and your responses to low memory warnings. This document points to you several memory management references that you might find useful.

Crash reports with stack traces need to be **symbolicated** before they can be analyzed. Symbolication replaces memory addresses with human-readable function names and line numbers. If you get crash logs off a device through Xcode's Organizer window, then they will be symbolicated for you automatically after a few seconds. Otherwise you will need to symbolicate the `.crash` file yourself by importing it to the Xcode Organizer. See Symbolication for details.

This document also talks about exception codes, another useful information for identifying the cause of the crash.

[Back to Top](#)

Understanding Low Memory Reports

When a low-memory condition is detected, the virtual memory system in iOS relies on the cooperation of applications to release memory. Low-memory notifications are sent to all running applications and processes as a request to free up memory, hoping to reduce the amount of memory in use. If memory pressure still exists, the system may terminate background processes to ease memory pressure. If enough memory can be freed up, your application will continue to run and no crash report will be

generated. If not, your application will be terminated by iOS because there isn't enough memory to satisfy the application's demands, and a low memory report will be generated and stored on the device.

The format of a low memory report differs from other crash reports in that there are no stack traces for the application threads. Memory usage of each process is reported in terms of number of memory pages, which as of this writing are 4KB each. You will see "(jettisoned)" next to the name of any process terminated by iOS to free up memory. If you see it next to your application's name, that confirms the application was terminated for using too much memory. Otherwise, the cause of the crash was not memory pressure. Look for a `.crash` file (described in the next section) for more information.

When you see a low memory crash, rather than be concerned about what part of your code was executing at the time of termination, you should investigate your memory usage patterns and your responses to low memory warnings. Memory Allocations Help lists detailed steps on how to use the Leaks Instrument to discover memory leaks, and how to use the Allocations Instrument's Mark Heap feature to avoid abandoned memory. Memory Usage Performance Guidelines discusses the proper ways to respond to low-memory notifications as well as many tips for using memory effectively. It is also recommended that you check out the WWDC 2010 session, Advanced Memory Analysis with Instruments.

Important: The Leaks and Allocations instruments do not track graphics memory. You need to run your application with the VM Tracker instrument (included in the Instruments Allocations template) to see your graphics memory usage. VM Tracker is disabled by default. To profile your application with VM Tracker, click the instrument, check the "Automatic Snapshotting" flag or press the "Snapshot Now" button manually.

[Back to Top](#)

Analyzing Crash Reports

Unlike the low memory reports, most crash reports contain stack traces for each thread at the time of termination. This section discusses these reports.

Symbolication

The most interesting part of a crash report is the stack trace of your application at the time execution halted. This trace is similar to what you would see when stopping execution in the debugger, except that you are not given the method or function names, known as symbols. Instead, you have hexadecimal addresses and executable code – your application or system frameworks – to which they refer. You need to map these addresses to symbols. The logs do not contain symbol information when they're written out. You have to symbolicate the logs before you can analyze them.

Symbolication – resolving stack trace addresses to source code methods and lines – requires the application binary that was uploaded to the App Store and the `.dSYM` file that was generated when that binary was built. This must be an exact match – otherwise, the report cannot be fully symbolicated. It is essential that you keep each build distributed to users (regardless of the details of that distribution) with its `.dSYM` file.

Important: You must keep both the application binary and the `.dSYM` file in order to be able to fully symbolicate crash reports. You should archive these files for every build that you submit to iTunes Connect. The `.dSYM` and application binary are specifically tied together on a per-build–

basis, and subsequent builds, even from the same source files, will not interoperate with files from other builds. If you use the “Build and Archive” command then they will be placed in a suitable location automatically. Otherwise any location searchable by Spotlight (such as your home directory) is fine.

Xcode's "Archive" command makes it easy keeping the matching binary and the `.dSYM`. When you use the "Archive" command (by choosing "Archive" from the "Product" menu or by pressing `Shift+Command+B`), Xcode will gather the application binary and the `.dSYM` containing symbol information together and store them in a location in your home folder. You can find all of your archived applications in the Xcode Organizer under the "Archived" section. Xcode will automatically search archived applications when symbolizing crash reports, and you can submit archived applications directly to iTunes Connect ensuring that the application and `.dSYM` that you have archived match what you release.

Xcode will automatically symbolicate all crash reports that it encounters, if it has the `.dSYM` and application binary that produced the crash report. Given a crash report, the matching binary, and its `.dSYM` file, all you need to do for symbolication is to add the crash report to the Xcode Organizer. Open the Xcode Organizer, select the “Devices” tab, select “Device Logs” under “LIBRARY” on the top of the sidebar, click the "Import" button and select the `.crash` file. When it's done, Xcode will automatically symbolicate the crash report and display the results.

Exception codes

In the crash log is a line that starts with the text `Exception Codes:` followed by one or more hexadecimal values. These are processor-specific codes that may give you more information on the nature of the crash.

- The exception code `0xbaaaaaad` indicates that the log is a stackshot of the entire system, not a crash report. To take a stackshot, push the home button and any volume button. Often these logs are accidentally created by users, and do not indicate an error.
- The exception code `0xbad22222` indicates that a VoIP application has been terminated by iOS because it resumed too frequently.
- The exception code `0x8badf00d` indicates that an application has been terminated by iOS because a watchdog timeout occurred. The application took too long to launch, terminate, or respond to system events. One common cause of this is doing synchronous networking on the main thread. Whatever operation is on `Thread 0:` needs to be moved to a background thread, or processed differently, so that it does not block the main thread.
- The exception code `0xc00010ff` indicates the app was killed by the operating system in response to a thermal event. This may be due to an issue with the particular device that this crash occurred on, or the environment it was operated in. For tips on making your app run more efficiently, see [iOS Performance and Power Optimization with Instruments WWDC session](#).
- The exception code `0xdead10cc` indicates that an application has been terminated by iOS because it held on to a system resource (like the address book database) while running in the background.
- The exception code `0xdeadfa11` indicated that an application has been force quit by the user. Force quits occur when the user first holds down the On/Off button until "slide to power off" appears, then holds down the Home button. It's reasonable to assume that the user has done this because the application has become unresponsive, but it's not guaranteed – force quit will work on any application.

Note: Terminating a suspended app by removing it from the multitasking tray does not generate a crash log. Once an app has suspended, it is eligible for termination by iOS at any time, so no crash log will be generated.

[Back to Top](#)

Related Documents

For information about how to acquire a crash report, see [Debugging Deployed iOS Apps](#).

For information about how to use the Instruments Zombies template to fix memory overrelease crashes, see [Finding Messages Sent To Deallocated Objects](#).

For more information about application archiving, see the [Distributing Applications](#) section of the Xcode 4 User Guide and [Testing Workflow with Xcode's Archive](#) feature.

For more information about interpreting crash logs, see [Understanding Crash Reports on iPhone OS WWDC 2010 Session](#).

[Back to Top](#)

Document Revision History

Date	Notes
2012-12-13	Added information about more exception codes.
2012-03-28	Added information about low memory crash reports and more exception codes. Updated for Xcode 4.
2011-03-01	Updated to reflect changes for iOS 4.0 and later.
2010-07-06	Fixed bug in the documentation.
2010-05-18	Updated to reflect changes for the iPhone OS 3.2 SDK and Xcode 3.2.2.
2009-06-01	Added stronger emphasis about the need to save not only .dSYM files, but application binaries as well.
2009-04-30	Updated for iTunes Connect crash log service.
2009-02-18	Updated to include a workaround for an issue that prevents application code from being symbolicated.
2009-01-29	New document that essential information for developers explaining how to symbolicate, understand, and interpret crash reports.
