Technical Note TN2239

iOS Debugging Magic

iOS contains a number of 'secret' debugging facilities, including environment variables, preferences, routines callable from GDB, and so on. This technote describes these facilities. If you're developing for iOS, you should look through this list to see if you're missing out on something that will make your life easier.

Introduction

All Apple systems include debugging facilities added by Apple engineering teams to help develop and debug specific subsystems. Many of these facilities remain in released system software and you can use them to debug your code. This technote describes some of the more broadly useful ones. In cases where a debugging facility is documented in another place, there's a short overview of the facility and a link to the existing documentation.

Important: This is not an exhaustive list: not all debugging facilities are, or will be, documented.

Many of the details covered in this technote vary from platform to platform and release to release. As such, you may encounter minor variations between platforms, and on older or newer systems. Known significant variations are called out in the text.

This technote was written with reference to iOS 4.1.

Warning: The debugging facilities described in this technote are unsupported. Apple reserves the right to change or eliminate each facility as dictated by the evolution of the OS; this has happened in the past, and is very likely to happen again in the future. **These facilities are for debugging only: you must not ship a product that relies on the existence or functionality of the facilities described in this technote.**

In addition to this technical issue of binary compatibility, keep in mind that iOS applications must comply with various legal agreements and the App Store Review Guidelines.

> **Note:** If you also develop for Mac OS X, you may want to read Technical Note TN2124, 'Mac OS X Debugging Magic'.

This technote covers advanced debugging techniques. If you're just getting started, you should consult the following material:

- GDB is the system's primary debugging tool. For a full description of GDB, see Debugging with GDB.
- Xcode is Apple's integrated development environment (IDE). It includes a sophisticated graphical debugger, implemented as a wrapper around GDB. For more information about Xcode, see the "Tools & Languages" section of the Apple developer Reference Library.

This technote does not cover performance debugging. If you're trying to debug a performance problem, the best place to start is the Getting Started with Performance document.

Back to Top

Basics

The later sections of this technote describe the debugging facilities in detail. Many of these facilities use similar techniques to enable and disable the facility, and to see its output. This section describes these common techniques.

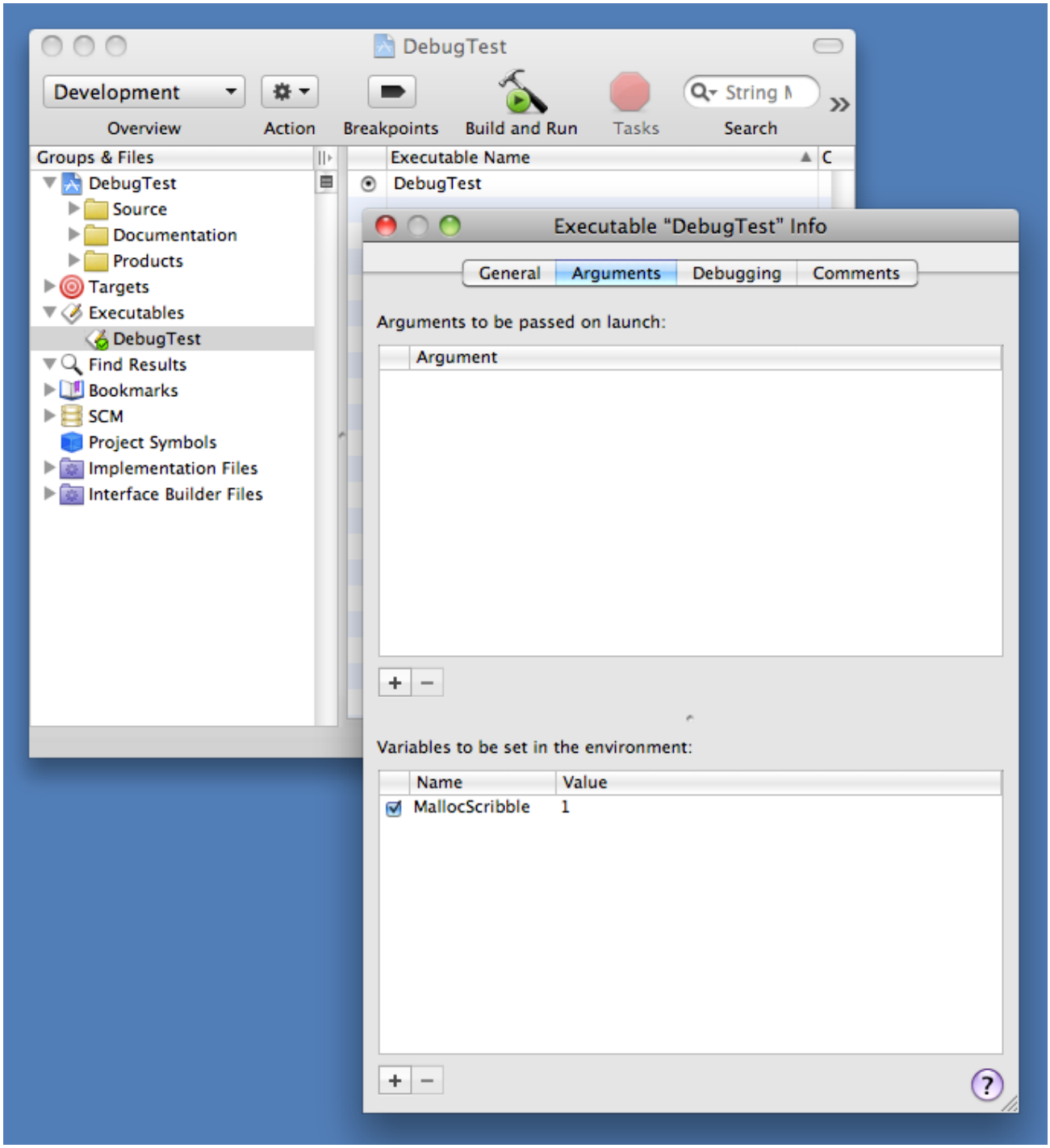
Enabling Debugging Facilities

Some debugging facilities are enabled by default. However, most facilities must be enabled using one of the techniques described in the following sections.

Environment Variables

In many cases you can enable a debugging facility by setting a particular environment variable. You can do this using the executable inspector in Xcode. Figure 1 shows an example of this.

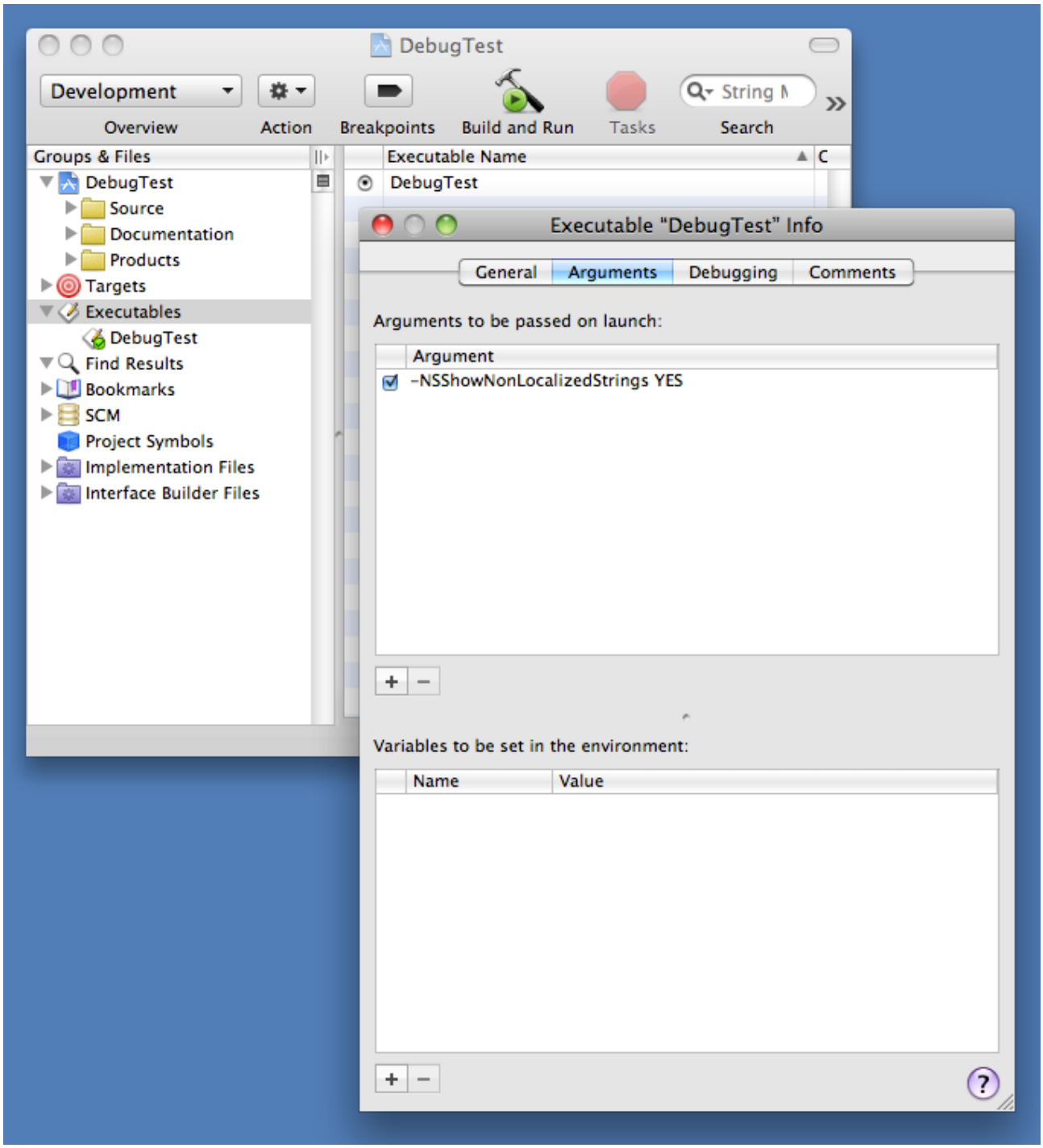
Figure 1: Setting environment variables in Xcode



Preferences

Some debugging facilities are enabled by setting a special preference. You can set such a debugging preference by configuring a command line argument in Xcode. Figure 2 shows how this is done.

Figure 2: Setting command line arguments in Xcode



Callable Routines

Many system frameworks include routines that print debugging information to `stderr`. These routines may be specifically designed to be callable from within GDB, or they may just be existing API routines that are useful while debugging. Listing 1 shows an example of how to call a debugging routine from GDB; specifically, it calls `CFBundleGetAllBundles` to get a list of all the bundles loaded in the application, and then prints that list by calling the `CFShow` routine.

Listing 1: Calling a debugging routine from GDB

```
(gdb) call (void)CFShow((void *)CFBundleGetAllBundles())
<CFArray 0x10025c010 [0x7fff701faf20]>{type = mutable-small, count = 59, values = (
    0 : CFBundle 0x100234d00 </System/Library/Frameworks/CoreData.framework> …
    […]
    12 : CFBundle 0x100237790 </System/Library/Frameworks/Security.framework> …
    […]
    23 : CFBundle 0x100194eb0 </System/Library/Frameworks/CoreFoundation.framework> …
    […]
)}
```

Note: When calling a routine like this, GDB must know the routine's return type (because some return types can affect the way in which parameters are passed). If you're calling a routine without debug symbols, you have to tell GDB the return type by adding a cast. For example, Listing 1 casts the return type of `CFBundleGetAllBundles` to `(void *)` and the return type of `CFShow` to `(void)`.

Similarly, if you call a routine with non-standard parameters, you may need to cast your arguments to ensure that GDB passes them correctly.

If you don't see the output from the routine, you may need to look at the console log, as described in the Seeing Debug Output section.

Important: If you use this technique for your own code, be warned that it doesn't always work for routines that are declared `static`. The compiler's interprocedural optimizations may cause a static routine to deviate from the standard function call ABI. In that case, it can't be reliably called by GDB.

In practice, this only affects Intel 32-bit code, as used by the iPhone Simulator.

Configuration Profiles

Some iOS subsystems have debugging facilities that can be enabled by installing a special configuration profile. For an example of this, see Push Notifications. You install such a configuration profile in the usual way:

- put it on a web server, and then download it using Safari on the device
- attach it to an email, mail it to an account accessible from the device, and then run Mail and open the configuration profile attachment

You can learn more about configuration profiles by reading the various documents accessible from iPhone in Business site.

Seeing Debug Output

Programs that generate debug output generally do so using one of following mechanisms:

- `NSLog`
- printing to `stderr`
- system log

`NSLog` is a high-level API for logging which is used extensively by Objective-C code. The exact behaviour of `NSLog` is surprisingly complex, and has changed significantly over time, making it beyond the scope of this document. However, it's sufficient to know that `NSLog` prints to `stderr`, or logs to the system log, or both. So, if you understand those two mechanisms, you can see anything logged via `NSLog`.

Printing to `stderr` is one of the most commonly used output mechanisms. Given this topic's importance, it is covered in depth in the next section.

The easiest way to view the system log is in the Console tab in Xcode's Organizer window. If you're working with users who don't want to install Xcode, they can view the system log using the iPhone Configuration Utility.

Console Output

Many programs, and indeed many system frameworks, print debugging messages to `stderr`. The destination for this output is ultimately controlled by the program: it can redirect `stderr` to whatever destination it chooses. However, in most cases a program does not redirect `stderr`, so the output goes to the default destination inherited by the program from its launch environment. This is typically one of the following:

- If you launch a GUI application as it would be launched by a normal user, the system redirects any messages printed on `stderr` to the system log. You can view these messages using the techniques described earlier.

- If you run a program from within Xcode, you can see its `stderr` output in Xcode's debugger Console window (choose the Console menu item from the Run menu to see this window).

Attaching to a running program (using Xcode's Attach to Process menu, or the `attach` command in GDB) does not automatically connect the program's `stderr` to your GDB window. You can do this from within GDB using the trick described in the "Seeing stdout and stderr After Attaching" section of Technical Note TN2030, 'GDB for MacsBug Veterans'.

Back to Top

Some Assembly Required

While it's very unusual to write a significant amount of code in assembly language these days, it's still useful to have a basic understanding of that dark art. This is particularly true when you're debugging, especially when you're debugging crashes that occur in libraries or frameworks for which you don't have the source code. This section covers some of the most basic techniques necessary to debug programs at the assembly level. Specifically, it describes how to set breakpoints, access parameters, and access the return address on all supported architectures.

Where the difference is significant the assembly-level examples in this technote are from an `armv7` binary running on an iPhone 3GS. However, in most cases the difference is not significant and the example might be from a variety of different architectures, perhaps even from the Mac. It's very easy to adapt such examples to other architectures. The most significant differences are:

- accessing parameters
- getting the return address

And these are exactly the items covered by the following architecture-specific sections.

> **Important:** The following architecture-specific sections contain rules of thumb. If the routine has any non-standard parameters, or a non-standard function result, these rules of thumb do not apply, and you should consult the documentation for the details.

In this context, **standard parameters** are integers (that fit in a single register), enumerations, and pointers (including pointers to arrays and pointers to functions). Non-standard parameters are floating point numbers, vectors, structures, integers bigger than a register, and any parameter after the last fixed parameter of a routine that takes a variable number of arguments.

For a detailed description of the calling conventions for all iOS devices, see iOS ABI Function Call Guide. The Intel 32-bit architecture used by the iPhone Simulator is described in the Mac OS X ABI Function Call Guide.

Before you read the following sections, it's critical that you understand one GDB subtlety. Because GDB is, at heart, a source-level debugger, when you set a breakpoint on a routine, GDB does not set the breakpoint on the first instruction of the routine; rather, it sets the breakpoint at the first instruction after the routine's prologue. From a source-level debugging point of view this make perfect sense. In a source-level debugger you never want to step through the routine's prologue. However, when doing assembly-level debugging, it's easier to access parameters before the prologue runs. That's because the location of the parameters at the first instruction of a routine is determined by the function call ABI, but the prologue is allowed to shuffle things around at its discretion. Moreover, each prologue can do this in a slightly different way. So the only way to access parameters after the prologue has executed is to disassemble the prologue and work out where everything went. This is typically, but not always, quite easy, but it's still extra work.

The best way to tell GDB to set a breakpoint at the first instruction of a routine is to prefix the routine name with a "*". Listing 2 shows an example of this.

Listing 2: Before and after the prologue

```
(gdb) b CFStringCreateWithFormat
Breakpoint 1 at 0x34427ff8
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x34427ff8 <CFStringCreateWithFormat+8>
-- The breakpoint is not at the first instruction.
-- Disassembling the routine shows that GDB has skipped the prologue.
(gdb) x/5i CFStringCreateWithFormat
0x34427ff0 <CFStringCreateWithFormat>: push    {r2, r3}
0x34427ff2 <CFStringCreateWithFormat+2>: push    {r7, lr}
```

```
0x34427ff4 <CFStringCreateWithFormat+4>: add r7, sp, #0
0x34427ff6 <CFStringCreateWithFormat+6>: sub sp, #4
0x34427ff8 <CFStringCreateWithFormat+8>: add r3, sp, #12
-- So we use a "*" prefix to disable GDB's 'smarts'.
(gdb) b *CFStringCreateWithFormat
Breakpoint 2 at 0x34427ff0
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x34427ff8 <CFStringCreateWithFormat+8>
2   breakpoint     keep n   0x34427ff0 <CFStringCreateWithFormat>
```

Important: Because iOS has no way to run programs from the command line, iOS-specific listings, like Listing 2, are taken from Xcode's Console window. And because the Console window does not support the '#' character for comments, comments are denoted by a leading '--'. If you try to replicate these examples yourself, you must not enter these lines into the Console window.

In contrast, some other listings in this document were created on Mac OS X, and thus use traditional GDB comments.

Finally, if you're looking for information about specific instructions, be aware that the Help menu in Shark (included in the Xcode developer tools) has an instruction set reference for ARM, Intel and PowerPC architectures.

ARM

In ARM programs the first four parameters are passed in registers. The return address is in register LR. Table 1 shows how to access these values from GDB when you've stopped at the first instruction of the function.

Table 1: Accessing parameters on ARM

What	GDB Syntax
return address	$lr
first parameter	$r0
second parameter	$r1
third parameter	$r2
fourth parameter	$r3

On return from a function the result is in register R0 (`$r0`).

Because parameters are passed in registers, there's no straightforward way to access parameters after the prologue.

Listing 3 shows an example of how to use this information to access parameters in GDB.

Listing 3: Parameters on ARM

```
-- We have to start the program from Xcode. Before we do that, we go to
-- Xcode's Breakpoints window and set a symbolic breakpoint on
-- CFStringCreateWithFormat.
GNU gdb 6.3.50-20050815 […]
-- We've stopped after the prologue.
(gdb) p/a $pc
$1 = 0x34427ff8 <CFStringCreateWithFormat+8>
-- Let's see what the prologue has done to the registers
-- holding our parameters.
(gdb) x/5i $pc-8
0x34427ff0 <CFStringCreateWithFormat>: push    {r2, r3}
0x34427ff2 <CFStringCreateWithFormat+2>: push    {r7, lr}
0x34427ff4 <CFStringCreateWithFormat+4>: add r7, sp, #0
```

```
0x34427ff6 <CFStringCreateWithFormat+6>: sub sp, #4
0x34427ff8 <CFStringCreateWithFormat+8>: add r3, sp, #12
-- Hey, the prologue hasn't modified R0..R3, so we're OK.
--
-- first parameter is "alloc"
(gdb) p/a $r0
$2 = 0x3e801d60 <__kCFAllocatorSystemDefault>
-- second parameter is "formatOptions"
(gdb) p/a $r1
$3 = 0x0
-- third parameter is "format"
(gdb) call (void)CFShow($r2)
managed/%@/%@
-- It turns out the prologue has left LR intact as well.
-- So we can get our return address.
--
-- IMPORTANT: Bit zero of the return address indicates that it lies
-- in a Thumb routine. So when using a return address in LR or on
-- the stack, always mask off bit zero.
(gdb) p/a $lr & 0xfffffffe
$4 = 0x34427e18 <__CFXPreferencesGetManagedSourceForBundleIDAndUser+48>
-- Now clear the breakpoint and set a new one before the prologue.
(gdb) del 1
(gdb) b *CFStringCreateWithFormat
Breakpoint 2 at 0x34427ff0
(gdb) c
Continuing.

Breakpoint 2, 0x34427ff0 in CFStringCreateWithFormat ()
-- We're at the first instruction. The parameters are guaranteed
-- to be in the right registers.
(gdb) p/a $pc
$5 = 0x34427ff0 <CFStringCreateWithFormat>
-- first parameter is "alloc"
(gdb) p/a $r0
$6 = 0x3e801d60 <__kCFAllocatorSystemDefault>
-- second parameter is "formatOptions"
(gdb) p/a $r1
$7 = 0x0
-- third parameter is "format"
(gdb) call (void)CFShow($r2)
%@/%@.plist
-- return address is in LR; again, we mask off bit zero
(gdb) p/a $lr & 0xfffffffe
$8 = 0x34427e7c <__CFXPreferencesGetManagedSourceForBundleIDAndUser+148>
(gdb) b *0x34427e7c
Breakpoint 3 at 0x34427e7c
-- Delete the other breakpoint to avoid thread-based confusion.
(gdb) del 2
-- Continue to the return address.
(gdb) c
Continuing.

Breakpoint 3, 0x34427e7c in __CFXPreferencesGetManagedSourceForBundleIDAndUser ()
-- function result
(gdb) p/a $r0
$9 = 0x1062d0
(gdb) call (void)CFShow($r0)
mobile/.GlobalPreferences.plist
```

Intel 32-Bit

The Intel 32-bit architecture is used in the iPhone Simulator.

In 32-bit Intel programs, parameters are passed on the stack. At the first instruction of a routine the top word of the stack contains the return address, the next word contains the first (leftmost) parameter, the next word contains the second parameter, and so on. Table 2 shows how to access these values from GDB.

Table 2: Accessing parameters on Intel 32-bit

What	GDB Syntax
return address	*(int*)$esp
first parameter	*(int*)($esp+4)

second parameter	*(int*)($esp+8)
... and so on	

After the routine's prologue you can access parameters relative to the frame pointer (register EBP). Table 3 shows the syntax.

Table 3: Accessing parameters after the prologue

What	GDB Syntax
previous frame	*(int*)$ebp
return address	*(int*)($ebp+4)
first parameter	*(int*)($ebp+8)
second parameter	*(int*)($ebp+12)
... and so on	

On return from a function the result is in register EAX (`$eax`).

Listing 4 shows an example of how to use this information to access parameters in GDB.

Listing 4: Parameters on Intel 32-Bit

```
$ # Use the -arch i386 argument to GDB to get it to run the
$ # 32-bit Intel binary.
$ gdb -arch i386 /Applications/TextEdit.app
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) fb CFStringCreateWithFormat
Breakpoint 1 at 0x31ec6d6
(gdb) r
Starting program: /Applications/TextEdit.app/Contents/MacOS/TextEdit
Reading symbols for shared libraries […]
Breakpoint 1, 0x940e36d6 in CFStringCreateWithFormat ()
(gdb) # We've stopped after the prologue.
(gdb) p/a $pc
$1 = 0x940e36d6 <CFStringCreateWithFormat+6>
(gdb) # However, for 32-bit Intel we don't need to inspect
(gdb) # the prologue because the parameters are on the stack.
(gdb) # We can access them relative to EBP.
(gdb) #
(gdb) # first parameter is "alloc"
(gdb) p/a *(int*)($ebp+8)
$2 = 0xa0473ee0 <__kCFAllocatorSystemDefault>
(gdb) # second parameter is "formatOptions"
(gdb) p/a *(int*)($ebp+12)
$3 = 0x0
(gdb) # third parameter is "format"
(gdb) call (void)CFShow(*(int*)($ebp+16))
%@
(gdb) # return address is at EBP+4
(gdb) p/a *(int*)($ebp+4)
$4 = 0x940f59fb <__CFXPreferencesGetNamedVolatileSourceForBundleID+59>
(gdb) # Now clear the breakpoint and set a new one before the prologue.
(gdb) del 1
(gdb) b *CFStringCreateWithFormat
Breakpoint 2 at 0x940e36d0
(gdb) c
Continuing.

Breakpoint 2, 0x940e36d0 in CFStringCreateWithFormat ()
(gdb) # We're at the first instruction. We must access
(gdb) # the parameters relative to ESP.
(gdb) p/a $pc
$6 = 0x940e36d0 <CFStringCreateWithFormat>
(gdb) # first parameter is "alloc"
```

```
(gdb) p/a *(int*)($esp+4)
$7 = 0xa0473ee0 <__kCFAllocatorSystemDefault>
(gdb) # second parameter is "formatOptions"
(gdb) p/a *(int*)($esp+8)
$8 = 0x0
(gdb) # third parameter is "format"
(gdb) call (void)CFShow(*(int*)($esp+12))
managed/%@/%@
(gdb) # return address is on the top of the stack
(gdb) p/a *(int*)$esp
$9 = 0x940f52cc <__CFXPreferencesGetManagedSourceForBundleIDAndUser+76>
(gdb) # Set a breakpoint on the return address.
(gdb) b *0x940f52cc
Breakpoint 3 at 0x940f52cc
(gdb) c
Continuing.

Breakpoint 3, 0x940f52cc in __CFXPreferencesGetManagedSourceForBundleIDAndUser ()
(gdb) # function result
(gdb) p/a $eax
$10 = 0x1079d0
(gdb) call (void)CFShow($eax)
managed/com.apple.TextEdit/kCFPreferencesCurrentUser
```

Architecture Gotchas

The following sections describe a couple of gotchas you might encounter when debugging at the assembly level.

Extra Parameters

When looking at parameters at the assembly level, keep in mind the following:

- If the routine is a C++ member function, there is an implicit first parameter for `this`.
- If the routine is an Objective-C method, there are two implicit first parameters (see Objective-C for details on this).
- If the compiler can find all the callers of a function (this most commonly happens when the function is declared as `static`), it can choose to pass parameters to that function in a non-standard way. This is very rare on architectures that have an efficient register-based ABI, but it's reasonably common for Intel 32-bit programs. So, if you set a breakpoint on a static function in an Intel 32-bit program, watch out for this very confusing behavior.

Endianness and Unit Sizes

When examining memory in GDB, things go smoother if you use the correct unit size. Table 4 is a summary of the unit sizes supported by GDB.

Table 4: GDB unit sizes

Size	C Type	GDB Unit	Mnemonic
1 byte	char	b	byte
2 bytes	short	h	half word
4 bytes	int	w	word
8 bytes	long or long long	g	giant

This is particularly important when debugging on little-endian systems (all iOS devices and the simulator) because you get confusing results if you specify the wrong unit size. Listing 5 shows an example of this (taken from the Mac, but the results would be the same on an iOS device). The second and third parameters of `CFStringCreateWithCharacters` specify an array of Unicode characters. Each element is a `UniChar`,

which is a 16-bit number in native-endian format. As we're running on a little-endian system, you have to dump this array using the correct unit size, otherwise everything looks messed up.

Listing 5: Using the right unit size

```
$ gdb
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) attach Finder
Attaching to process 4732.
Reading symbols for shared libraries . done
Reading symbols for shared libraries […]
0x00007fff81963e3a in mach_msg_trap ()
(gdb) b *CFStringCreateWithCharacters
Breakpoint 1 at 0x7fff86070520
(gdb) c
Continuing.

Breakpoint 1, 0x00007fff86070520 in CFStringCreateWithCharacters ()
(gdb) # The third parameter is the number of UniChars
(gdb) # in the buffer pointed to by the first parameter.
(gdb) p (int)$rdx
$1 = 18
(gdb) # Dump the buffer as shorts. Everything makes sense.
(gdb) # This is the string "Auto-Save Recovery".
(gdb) x/18xh $rsi
0x10b7df292: 0x0041  0x0075  0x0074  0x006f  0x002d  0x0053  0x0061  0x0076
0x10b7df2a2: 0x0065  0x0020  0x0052  0x0065  0x0063  0x006f  0x0076  0x0065
0x10b7df2b2: 0x0072  0x0079
(gdb) # Now dump the buffer as words. Most confusing.
(gdb) # It looks like "uAotS-va eeRocevyr"!
(gdb) x/9xw $rsi
0x10b7df292: 0x00750041      0x006f0074      0x0053002d      0x00760061
0x10b7df2a2: 0x00200065      0x00650052      0x006f0063      0x00650076
0x10b7df2b2: 0x00790072
(gdb) # Now dump the buffer as bytes. This is a little less
(gdb) # confusing, but you still have to remember that it's big
(gdb) # endian data.
(gdb) x/36xb $rsi
0x10b7df292: 0x41    0x00    0x75    0x00    0x74    0x00    0x6f    0x00
0x10b7df29a: 0x2d    0x00    0x53    0x00    0x61    0x00    0x76    0x00
0x10b7df2a2: 0x65    0x00    0x20    0x00    0x52    0x00    0x65    0x00
0x10b7df2aa: 0x63    0x00    0x6f    0x00    0x76    0x00    0x65    0x00
0x10b7df2b2: 0x72    0x00    0x79    0x00
```

Controlled Crash

In some cases it's useful to crash your program in a controlled manner. One common way to do this is to call abort. Another option is to use the `__builtin_trap` intrinsic function, which generates a machine-specific trap instruction. Listing 6 shows how this is done.

Listing 6: Crashing via __builtin_trap

```
int main(int argc, char **argv) {     __builtin_trap();     return 1; }
```

Note: The `__builtin_trap` intrinsic function generates a `trap` instruction on PowerPC and ARM, and a `ud2a` instruction on Intel.

If you run the program in the debugger, you will stop at the line immediately before the `__builtin_trap` call. Otherwise the program will crash and generate a crash report.

Warning: We recommend that you restrict this technique to debug builds; your release builds should use `abort`.

Be mindful that the iOS application lifecycle is under user control, meaning that iOS applications should not just quit. Your release build should only call `abort` in circumstances where it would have crashed anyway, and the `abort` call prevents damage to user data or allows you to more easily diagnose the problem.

Back to Top

Instruments

Instruments is an application for dynamically tracing and profiling code. It runs on Mac OS X and allows you to target programs running on Mac OS X, iOS devices, and the iPhone Simulator.

While Instruments is primarily focused on performance debugging, you can also use it for debugging errors. For example, the ObjectAlloc instrument can help you track down both over- and under-release bugs.

One particularly nice feature of Instruments is that it gives you easy access to zombies. See the Instruments User Guide for details on this and other Instruments features.

Back to Top

CrashReporter

CrashReporter is an invaluable debugging facility that logs information about all programs that crash. It is enabled all the time; all you have to do is look at its output.

CrashReporter is described in detail in Technical Note TN2151, 'Understanding and Analyzing iPhone OS Application Crash Reports'.

Back to Top

BSD

The BSD subsystem implements process, memory, file, and network infrastructure, and thus is critical to all programs on the system. BSD implements a number of neat debugging facilities that you can take advantage of.

Memory Allocator

The default memory allocator includes a number of debugging facilities that you can enable via environment variables. These are fully documented in the manual page. Table 5 lists some of the more useful ones.

Table 5: Some useful memory allocator environment variables

Variable	Summary
MallocScribble	Fill allocated memory with 0xAA and scribble deallocated memory with 0x55
MallocGuardEdges	Add guard pages before and after large allocations
MallocStackLogging	Record backtraces for each memory block to assist memory debugging tools; if the block is allocated and then immediately freed, both entries are removed from the log, which helps reduce the size of the log
MallocStackLoggingNoCompact	Same as MallocStackLogging but keeps all log entries

The default memory allocator also logs messages if it detects certain common programming problems. For example, if you free a block of memory twice, or free memory that you never allocated, `free` may print the message shown in Listing 7. The number inside parentheses is the process ID.

Listing 7: A common message printed by free

```
DummyPhone(1839) malloc: *** error for object 0x826600: double free *** set a breakpoint
in malloc_error_break to debug
```

You can debug this sort of problem by running your program within GDB and putting a breakpoint on `malloc_error_break`. Once you hit the breakpoint, you can use GDB's `backtrace` command to determine the immediate caller.

Finally, you can programmatically check the consistency of the heap using the `malloc_zone_check` routine (from `<malloc.h>`).

Standard C++ Library

The standard C++ library supports a number of debugging features:

- Set the `_GLIBCXX_DEBUG` compile-time variable to enable debug mode in the standard C++ library.

  This item is not supported in GCC 4.2 or later, and you must not use it in that context.

- In versions prior to GCC 4.0, set the `GLIBCPP_FORCE_NEW` environment variable to 1 to disable memory caching in the standard C++ library. This allows you to debug your C++ allocations with the other memory debugging tools.

  In GCC 4.0 and later this is the default behavior.

Dynamic Linker (dyld)

The dynamic linker (dyld) supports a number of debugging facilities that you can enable via environment variables. These are fully documented in the manual page. Table 6 lists some of the more useful variables.

Table 6: Dynamic linker environment variables

Variable	Summary
DYLD_IMAGE_SUFFIX	Search for libraries with this suffix first
DYLD_PRINT_LIBRARIES	Log library loads
DYLD_PRINT_LIBRARIES_POST_LAUNCH	As above, but only after main has run
DYLD_PRINT_OPTS [1]	Print launch-time command line arguments
DYLD_PRINT_ENV [1]	Print launch-time environment variables
DYLD_PRINT_APIS [1]	Log dyld API calls (for example, dlopen)
DYLD_PRINT_BINDINGS [1]	Log symbol bindings
DYLD_PRINT_INITIALIZERS [1]	Log image initialization calls
DYLD_PRINT_SEGMENTS [1]	Log segment mapping
DYLD_PRINT_STATISTICS [1]	Print launch performance statistics

Notes:

1. On the Mac OS X side these are only available on Mac OS X 10.4 and later. They are, however, available on all versions of iOS.

While these environment variables are implemented on iOS, many of them have limited utility because of the restricted environment on that system.

Back to Top

Core Services

Core Foundation

The Core Foundation (CF) framework exports the `CFShow` routine, which prints a description of any CF object to `stderr`. You can make this call from your own code, however, it's particularly useful when called from GDB. Listing 8 shows an example of this.

Listing 8: Calling CFShow from GDB

```
$ gdb /Applications/TextEdit.app
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) fb CFRunLoopAddSource
Breakpoint 1 at 0x624dd2f195cfa8
(gdb) r
Starting program: /Applications/TextEdit.app/Contents/MacOS/TextEdit
Reading symbols for shared libraries […]
Breakpoint 1, 0x00007fff8609bfa8 in CFRunLoopAddSource ()
(gdb) # Check that the prologue hasn't changed $rdi.
(gdb) p/a 0x00007fff8609bfa8
$1 = 0x7fff8609bfa8 <CFRunLoopAddSource+24>
(gdb) p/a $pc
$2 = 0x7fff8609bfa8 <CFRunLoopAddSource+24>
(gdb) x/8i $pc-24
0x7fff8609bf90 <CFRunLoopAddSource>: push   %rbp
0x7fff8609bf91 <CFRunLoopAddSource+1>: mov    %rsp,%rbp
0x7fff8609bf94 <CFRunLoopAddSource+4>: mov    %rbx,-0x20(%rbp)
0x7fff8609bf98 <CFRunLoopAddSource+8>: mov    %r12,-0x18(%rbp)
0x7fff8609bf9c <CFRunLoopAddSource+12>: mov    %r13,-0x10(%rbp)
0x7fff8609bfa0 <CFRunLoopAddSource+16>: mov    %r14,-0x8(%rbp)
0x7fff8609bfa4 <CFRunLoopAddSource+20>: sub    $0x40,%rsp
0x7fff8609bfa8 <CFRunLoopAddSource+24>: mov    %rdi,%r12
(gdb) # Nope. Go ahead and CFShow it.
(gdb) call (void)CFShow($rdi)
<CFRunLoop 0x100115540 [0x7fff70b8bf20]>{
    locked = false,
    wakeup port = 0x1e07,
    stopped = false,
    current mode = (none),
    common modes = <CFBasicHash 0x1001155a0 [0x7fff70b8bf20]>{
        type = mutable set,
        count = 1,
        entries =>
            2 : <CFString 0x7fff70b693d0 [0x7fff70b8bf20]>{
                contents = "kCFRunLoopDefaultMode"
            }
    },
    common mode items = (null),
    modes = <CFBasicHash 0x1001155d0 [0x7fff70b8bf20]>{
        type = mutable set,
        count = 1,
        entries =>
            0 : <CFRunLoopMode 0x100115670 [0x7fff70b8bf20]>{
                name = kCFRunLoopDefaultMode,
                locked = false,
                port set = 0x1f03,
                sources = (null),
                observers = (null),
                timers = (null)
        },
    }
}
```

Important: If you don't see any output from `CFShow`, it was probably sent to the console. See Seeing Debug Output for information on how to view this output.

Note: In Listing 8 the output from `CFShow` has been reformatted to make it easier to read.

There are a number of other CF routines that you might find useful to call from GDB, including `CFGetRetainCount`, `CFBundleGetMainBundle`, and `CFRunLoopGetCurrent`.

Zombies!

Important: If you're programming in Objective-C, you're more likely to be interested in `NSZombieEnabled`, as described in More Zombies!.

Core Foundation supports an environment variable called `CFZombieLevel`. It interprets this variable as an integer containing a set of flag bits. Table 7 describes the bits that are currently defined. These can help you track down various CF memory management issues.

Table 7: Bit definitions for CFZombieLevel environment variable

Bit	Action
0	scribble deallocated CF memory
1	when scribbling deallocated CF memory, don't scribble object header (CFRuntimeBase)
4	never free memory used to hold CF objects
7	if set, scribble deallocations using bits 8..15, otherwise use 0xFC
8..15	if bit 7 is set, scribble deallocations using this value
16	scribble allocated CF memory
23	if set, scribble allocations using bits 24..31, otherwise use 0xCF
24..31	if bit 23 is set, scribble allocations using this value

Back to Top

Application Services

Core Animation

The Core Animation instrument lets you measure your application's frame rate and see various types of drawing. See Instruments User Guide for details.

Back to Top

Cocoa and Cocoa Touch

All Cocoa objects (everything derived from `NSObject`) support a `description` method that returns an `NSString` describing the object. The most convenient way to access this description is via Xcode's Print Description to Console menu command. Alternatively, if you're a command line junkie, you can use GDB's `print-object` (or `po` for short) command, as illustrated by Listing 9.

Listing 9: Using GDB's po command

```
$ gdb /Applications/TextEdit.app
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) fb -[NSCFDictionary copyWithZone:]
Breakpoint 1 at 0x83126e97675259
(gdb) r
Starting program: /Applications/TextEdit.app/Contents/MacOS/TextEdit
Reading symbols for shared libraries […]
Breakpoint 1, 0x00007fff837aa259 in -[NSCFDictionary copyWithZone:] ()
(gdb) po $rdi
{
    AddExtensionToNewPlainTextFiles = 1;
```

```
    AutosaveDelay = 30;
    CheckGrammarWithSpelling = 0;
    CheckSpellingWhileTyping = 1;
    […]
}
```

Objective-C

To break on an Objective-C exception, regardless of how it's thrown, set a symbolic breakpoint on `objc_exception_throw`. The easiest way to set such a breakpoint is with Xcode's Stop on Objective-C Exceptions menu command.

Note: This is better than setting a breakpoint on `-[NSException raise]` because it will be hit even if the exception is thrown using `@throw`.

Assembly-Level Objective-C Debugging

When debugging Cocoa code at the assembly level, keep in mind the following features of the Objective-C runtime:

- The Objective-C compiler adds two implicit parameters to each method, the first of which is a pointer to the object being called (`self`).
- The second implicit parameter is the method selector (`_cmd`). In Objective-C this is of type `SEL`; in GDB you can print this as a C string.
- The Objective-C runtime dispatches methods via a family of C function. The most commonly seen is `objc_msgSend`, but some architectures use `objc_msgSend_stret` for methods which returns structures, and some architectures use `objc_msgSend_fpret` for methods that return floating point values. There are also equivalent functions for calling `super` (`objc_msgSendSuper` and so on).
- The first word of any Objective-C object (the `isa` field) is a pointer to the object's class.

Note: If you're interested in learning more about Objective-C message dispatch, you should check out this article.

Table 8 is a summary of how to access `self` and `_cmd` from GDB if you've stopped at the first instruction of a method. For more information about this, see Some Assembly Required.

Table 8: Accessing self and _cmd

Architecture	self	_cmd
ARM	$r0	$r1
Intel 32-bit	*(int*)($esp+4)	*(int*)($esp+8)

Listing 10 shows an example of how to use this information from GDB.

Important: Listing 10 and Listing 11 were created on Mac OS X, but the core techniques work the same on iOS; read Some Assembly Required for information on how to translate these techniques to iOS.

Listing 10: Objective-C runtime 'secrets'

```
$ gdb /Applications/TextEdit.app
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) # Give the runtime a chance to start up.
(gdb) fb NSApplicationMain
```

```
Breakpoint 1 at 0x9374bc69df7307
(gdb) r
Starting program: /Applications/TextEdit.app/Contents/MacOS/TextEdit
Reading symbols for shared libraries […]
Breakpoint 1, 0x00007fff841a0307 in NSApplicationMain ()
(gdb) # Set a breakpoint on -retain.
(gdb) b *'-[NSObject(NSObject) retain]'
Breakpoint 2 at 0x7fff8608a860
(gdb) c
Continuing.

Breakpoint 2, 0x00007fff8608a860 in -[NSObject(NSObject) retain] ()
(gdb) # Hit the breakpoint; dump the first 4 words of the object
(gdb) x/4xg $rdi
0x1001138f0: 0x00007fff7055e6d8  0x0041002f01a00000
0x100113900: 0x0069006c00700070  0x0069007400610063
(gdb) # Now print the selector
(gdb) x/s $rsi
0x7fff848d73d8: "retain"
(gdb) # Want to 'po' object; must disable the breakpoint first
(gdb) dis
(gdb) po $rdi
/Applications/TextEdit.app
(gdb) # Print the 'isa' pointer, which is a Class object.
(gdb) po 0xa02d9740
NSPathStore2
```

When debugging without symbols, you can use functions from the Objective-C runtime to assist your debugging efforts. The routines shown in Table 9 are particularly useful.

Table 9: Useful Objective-C runtime functions

Function	Summary
id objc_getClass(const char *name);	Get the Objective-C Class object for the given class name
SEL sel_getUid(const char *str);	Get the Objective-C SEL for the given method name
IMP class_getMethodImplementation(Class cls, SEL name);	Get a pointer to the code that implements the given method in a given class

Listing 11 shows an example of debugging the `-[DocumentController openUntitledDocumentAndDisplay:error:]` method of TextEdit, even though TextEdit does not ship with symbols.

Listing 11: Using the Objective-C runtime to debug without symbols

```
$ gdb -arch x86_64 /Applications/TextEdit.app
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) […]
(gdb) r
Starting program: /Applications/TextEdit.app/Contents/MacOS/TextEdit
Reading symbols for shared libraries […]
^C
(gdb) # Try to find the
(gdb) # -[DocumentController openUntitledDocumentAndDisplay:error:]
(gdb) # symbol.
(gdb) info func openUntitledDocumentAndDisplay
All functions matching regular expression "openUntitledDocumentAndDisplay":

Non-debugging symbols:
0x00007fff843ac083 -[NSDocumentController openUntitledDocumentAndDisplay:error:]
(gdb) # These are not the droids we're looking for. It turns out that
(gdb) # TextEdit ships with its symbols stripped, so we'll have to do
(gdb) # this the hard way.
(gdb) #
(gdb) # Get the Class object for the DocumentController class.
(gdb) set $class=(void *)objc_getClass("DocumentController")
(gdb) # Get the SEL object for the "openUntitledDocumentAndDisplay:error:" method.
(gdb) set $sel=(void *)sel_getUid("openUntitledDocumentAndDisplay:error:")
(gdb) # Get a pointer to the method implementation.
(gdb) call (void*)class_getMethodImplementation($class, $sel)
$1 = (void *) 0x100001966
(gdb) # Confirm that this is sensible. Looks like a method prologue to me.
(gdb) x/4i 0x00009aa5
```

```
0x100001966: push   %rbp
0x100001967: mov    %rsp,%rbp
0x10000196a: push   %r12
0x10000196c: push   %rbx
(gdb) # Set a breakpoint on the method.
(gdb) b *0x100001966
Breakpoint 1 at 0x100001966
(gdb) # Resume execution, and then create a new, untitled document.
(gdb) c
Continuing.
[…]
Breakpoint 1, 0x0000000100001966 in ?? ()
(gdb) # We've hit our breakpoint; print the parameters, starting with
(gdb) # the implicit "self" and "SEL" parameters that are common to all
(gdb) # methods, followed by the method-specific "display" and
(gdb) # "error" parameters.
(gdb) po $rdi
<DocumentController: 0x100227a50>
(gdb) x/s $rsi
0x7fff848e4e04: "openUntitledDocumentAndDisplay:error:"
(gdb) p (int)$rdx
$2 = 1
(gdb) x/xg $rcx
0x7fff5fbff108: 0x00000001001238f0
```

You can learn more about the Objective-C runtime functions and data structures by looking through the headers in `/usr/include/objc/`.

> **Warning:** The Objective-C runtime lets you uncover many private implementation details of system classes. You must not use any of this information in your final product.

Foundation

Foundation has a number of debugging facilities that are enabled by environment variables. Table 10 highlights the most interesting ones.

Table 10: Foundation environment variables

Name	Default	Action
NSZombieEnabled	NO	If set to YES, deallocated objects are 'zombified'; this allows you to quickly debug problems where you send a message to an object that has already been freed; see More Zombies! for details
NSDeallocateZombies	NO	If set to YES, the memory for 'zombified' objects is actually freed
NSUnbufferedIO	NO	If set to YES, Foundation will use unbuffered I/O for stdout (stderr is unbuffered by default)

> **Important:** To enable or disable a Foundation debugging facility, you must set the value of the environment variable to "YES" or "NO", not 1 or 0 as is the case with other system components.

Retain Counts

You can use `-retainCount` to get the current retain count of an object. While this can sometimes be a useful debugging aid, be very careful when you interpret the results. Listing 12 shows one potential source of confusion.

Listing 12: Confusing retain count

```
(gdb) set $s=(void *)[NSClassFromString(@"NSString") string]
(gdb) p (int)[$s retainCount]
```

```
$4 = 2147483647
(gdb) p/x 2147483647
$5 = 0x7fffffff
(gdb) # The system maintains a set of singleton strings for commonly
(gdb) # used values, like the empty string. The retain count for these
(gdb) # strings is a special value indicating that the object can't be
(gdb) # released.
```

Another common source of confusion is the autorelease mechanism. If an object has been autoreleased, its retain count is higher than you might otherwise think, a fact that's compensated for by the autorelease pool releasing it at some point in the future.

You can determine what objects are in what autorelease pools by calling `_CFAutoreleasePoolPrintPools` to print the contents of all the autorelease pools on the autorelease pool stack.

Listing 13: Printing the autorelease pool stack

```
(gdb) call (void)_CFAutoreleasePoolPrintPools()
- -- ---- -------- Autorelease Pools -------- ---- -- -
==== top of stack ================
  0x327890 (NSCFDictionary)
  0x32cf30 (NSCFNumber)
  […]
==== top of pool, 10 objects ================
  0x306160 (__NSArray0)
  0x127020 (NSEvent)
  0x127f60 (NSEvent)
==== top of pool, 3 objects ================
- -- ---- -------- ----------------- -------- ---- -- -
```

Note: `_CFAutoreleasePoolPrintPools` is only available on iOS 4.0 or later.

On earlier systems you can use the `NSAutoreleasePoolCountForObject` function to determine how many times an object has been added to an autorelease pool. For an example, see Listing 14.

Listing 14: Calling NSAutoreleasePoolCountForObject

```
(gdb) # d is an NSDictionary created with -[NSDictionary dictionaryWithObjectsAndKeys:].
(gdb) p d
$1 = (NSDictionary *) 0x12d620
(gdb) po d
{
    test = 12345;
}
(gdb) p (int)[d retainCount]
$2 = 1
(gdb) p (int)NSAutoreleasePoolCountForObject(d)
$3 = 1
```

More Zombies!

A common type of bug when programming with Cocoa is over-releasing an object. This typically causes your application to crash, but the crash occurs after the last reference count is released (when you try to message the freed object), which is usually quite removed from the original bug. `NSZombieEnabled` is your best bet for debugging this sort of problems; it will uncover any attempt to interact with a freed object.

The easiest way to enable zombies is via Instruments. However, you can also enable zombies via an environment variable. Listing 15 shows the type of the message you'll see in this case.

Listing 15: The effect of NSZombie

```
$ NSZombieEnabled=YES build/Debug/DummyMac.app/Contents/MacOS/DummyMac  […] -[AppDelegate
testAction:] […] *** -[CFNumber release]: message sent to deallocated instance 0x3737c0
Trace/BPT trap
```

As shown in Listing 15, the system will execute a breakpoint instruction if it detects a zombie. If you're running under GDB, the program will stop and you can look at the backtrace to see the chain of calls that triggered the zombie detector.

`NSZombieEnabled` also affects Core Foundation objects as well as Objective-C objects. However, you'll only be notified when you access a zombie Core Foundation object from Objective-C, not when you access it via a CF API.

Other Foundation Magic

If you're using Key-Value Observing and you want to know who is observing what on a particular object, you can get the observation information for that object and print it using using GDB's print-object command. Listing 16 shows an example of this.

Listing 16: Display key-value observers

```
(gdb) # self is some Objective-C object.
(gdb) po self
<ZoneInfoManager: 0x48340d0>
(gdb) # Let's see who's observing what key paths.
(gdb) po [self observationInfo]
<NSKeyValueObservationInfo 0x48702d0> (
<NSKeyValueObservance 0x4825490: Observer: 0x48436e0, \
Key path: zones, Options: <New: NO, Old: NO, Prior: NO> \
Context: 0x0, Property: 0x483a820>
)
```

You can set the `NSShowNonLocalizedStrings` preference to find strings that should have been localized but weren't. Once enabled, if you request a localized string and the string is not found in a strings file, the system will return the string capitalized and log a message to the console. This is a great way to uncover problems with out-of-date localizations.

This setting affects `NSLocalizedString`, all of its variants, and all underlying infrastructure including `CFCopyLocalizedString`.

UIKit

UIView implements a useful `description` method. In addition, it implements a `recursiveDescription` method that you can call to get a summary of an entire view hierarchy.

Listing 17: UIView's description and recursiveDescription methods

```
-- The following assumes that you're stopped in some view controller method.
(gdb) po [self view]
<UIView: 0x6a107c0; frame = (0 20; 320 460); autoresize = W+H; layer = […]
Current language: auto; currently objective-c
(gdb) po [[self view] recursiveDescription]
<UIView: 0x6a107c0; frame = (0 20; 320 460); autoresize = W+H; layer = […]
   | <UIRoundedRectButton: 0x6a103e0; frame = (124 196; 72 37); opaque = NO; […]
   |    | <UIButtonLabel: 0x6a117b0; frame = (19 8; 34 21); text = 'Test'; […]
```

Back to Top

Networking

The most critical tool for debugging network code is the packet trace. Technical Q&A QA1176, 'Getting a Packet Trace' discusses how to get a packet trace on Mac OS X.

While iOS has no built-in packet tracing facilities, you can often make useful progress by taking your packet trace on a nearby Mac.

Back to Top

Power

The Energy Diagnostics instrument is a great way to understand how your application affects battery life. See Instruments User Guide for details.

Back to Top

Push Notifications

Technical Note TN2265, 'Troubleshooting Push Notifications' is a comprehensive guide to debugging push notification problems. It also includes a configuration profile (`APNsLogging.mobileconfig`) that you can install to enable additional debug logging.

Back to Top

iPhone Simulator

iPhone Simulator lets your build and run your iOS code on Mac OS X. Because iPhone Simulator is a Mac OS X application, you can target it with a variety of Mac OS X specific debugging facilities. For example, if you have a problem that can only be solved by DTrace (which is not available on iOS), you can run your application in the simulator and point DTrace at that.

> **Important:** The simulator is great for debugging, but **the ultimate arbiter of what will and won't work on iOS is a real device**. It is especially important to keep this in mind when doing performance testing and debugging.

For more information about Mac OS X debugging facilities, see Technical Note TN2124, 'Mac OS X Debugging Magic'.

Back to Top

Document Revision History

Date	Notes
2011-01-22	New document that describes a large collection of iOS debugging hints and tips.
