

iOS ABI Function Call Guide

Contents

Introduction 4

Organization of This Document 4

ARM64 Function Calling Conventions 5

Choices Made Within the Generic Procedure Call Standard 5

Divergences from the Generic Procedure Call Standard 5

Argument Passing in General 5

Variadic Functions 6

Fundamental C Types 6

Red Zone 7

Divergences from the Generic C++ ABI 7

Name Mangling 7

Other Itanium Divergences 7

Data Types and Data Alignment 8

ARMv7 Function Calling Conventions 10

Function Calls 10

Prologs and Epilogs 10

Register Preservation 11

ARMv6 Function Calling Conventions 12

Data Types and Data Alignment 12

Function Calls 14

Calling a Function 14

Stack Structure 15

Prologs and Epilogs 16

Passing Arguments 18

Returning Results 18

Register Preservation 18

Document Revision History 21

Figures, Tables, and Listings

ARM64 Function Calling Conventions 5

- Table 1 Size and alignment of integer data types 8
- Listing 1 Example of space occupied by values 6
- Listing 2 Example of 16-bit aligned arguments passed in integer registers 6

ARMv7 Function Calling Conventions 10

- Table 1 Additional processor registers in the ARMv7 architecture 11
- Listing 1 Example prolog for ARM or Thumb-2 (ARMv7) 10
- Listing 2 Example epilog for ARM or Thumb-2 (ARMv7) 11

ARMv6 Function Calling Conventions 12

- Figure 1 Stack layout 15
- Table 1 Size and natural alignment of the scalar data types 13
- Table 2 Processor registers in the ARMv6 architecture 18
- Listing 1 Example prolog for ARM (ARMv6) 16
- Listing 2 Example epilog for ARM (ARMv6) 17
- Listing 3 Example prolog for Thumb (ARMv6) 17
- Listing 4 Example epilog for Thumb (ARMv6) 17

Introduction

This document describes the function-calling conventions used in the iOS ABI on the architectures on which iOS can run. Specifically, this document covers the ARM64 architecture, as well as the v7 and v6 revisions of the ARM 32-bit architecture.

The information in this document is based on iOS 2.0 and later, and Xcode 3.1 and later.

This document is intended for developers interested in the calling conventions used in the iOS ABI on each of the supported architectures. This information is especially useful to developers of development tools.

Organization of This Document

This document contains the following articles:

- [“ARM64 Function Calling Conventions”](#) (page 5)
- [“ARMv7 Function Calling Conventions”](#) (page 10)
- [“ARMv6 Function Calling Conventions”](#) (page 12)

Each of these articles describes how routines pass arguments to the functions they call, how functions pass results to their callers, and the data types that can be used to manipulate the arguments and results of function calls.

ARM64 Function Calling Conventions

In general, iOS adheres to the generic ABI specified by ARM for the ARM64 architecture. However there are some choices to be made within that framework, and some divergences from it. This document describes these issues.

Choices Made Within the Generic Procedure Call Standard

[Procedure Call Standard for the ARM 64-bit Architecture](#) delegates certain decisions to platform designers. Decisions made for iOS are described below.

- The register x18 is reserved for the platform. Conforming software should not make use of it.
- `wchar_t` is 32-bit and `long` is a 64-bit type.
- Where applicable, the `__fp16` type is IEEE754-2008 format.
- The frame pointer register (x29) must always address a valid frame record, although some functions—such as leaf functions or tail calls—may elect not to create an entry in this list. As a result, stack traces will always be meaningful, even without debug information.
- Empty struct types are ignored for parameter-passing purposes. This behavior applies to the GNU extension in C and, where permitted by the language, in C++. (This issue is not directly specified by the generic procedure call standard, but a decision was required.)

Divergences from the Generic Procedure Call Standard

iOS diverges from [Procedure Call Standard for the ARM 64-bit Architecture](#) in several ways, as described here.

Argument Passing in General

- In the generic procedure call standard, all function arguments passed on the stack consume slots in multiples of 8 bytes. In iOS, this requirement is dropped, and values consume only the space required. For example, on entry to the function in Listing 1, `s0` occupies 1 byte at `sp` and `s1` occupies 1 byte at `sp+1`. Padding is still inserted on the stack to satisfy arguments' alignment requirements.

Listing 1 Example of space occupied by values

```
void two_stack_args(char w0, char w1, char w2, char w3, char w4, char w5,  
char w6, char w7, char s0, char s1) {}
```

- The generic procedure call standard requires that arguments with 16-byte alignment passed in integer registers begin at an even-numbered xN, skipping a previous odd-numbered xN if necessary. The iOS ABI drops this requirement. For example, in Listing 2, the parameter x1_x2 does indeed get passed in x1 and x2 instead of x2 and x3.

Listing 2 Example of 16-bit aligned arguments passed in integer registers

```
void large_type(int x0, __int128 x1_x2) {}
```

- The general ABI specifies that it is the callee's responsibility to sign or zero-extend arguments having fewer than 32 bits, and that unused bits in a register are unspecified. In iOS, however, the caller must perform such extensions, up to 32 bits.

Variadic Functions

The iOS ABI for functions that take a variable number of arguments is entirely different from the generic version.

Stages A and B of the generic procedure call standard are performed as usual—in particular, even variadic aggregates larger than 16 bytes are passed via a reference to temporary memory allocated by the caller. After that, the fixed arguments are allocated to registers and stack slots as usual in iOS.

The NSRN is then rounded up to the next multiple of 8 bytes, and each variadic argument is assigned to the appropriate number of 8-byte stack slots.

The C language requires arguments smaller than `int` to be promoted before a call, but beyond that, unused bytes on the stack are not specified by this ABI.

As a result of this change, the type `va_list` is an alias for `char *` rather than for the struct type specified in the generic PCS. It is also not in the `std` namespace when compiling C++ code.

Fundamental C Types

The iOS version of the ABI has the following differences from the generic ABI in the fundamental types provided by the C language.

- Generally, `long double` is a quad-precision IEEE754 binary floating-point type. In iOS, however, it is a double-precision IEEE754 binary floating-point type. In other words, `long double` is identical to `double` in iOS.
- In iOS, as with other Darwin platforms, both `char` and `wchar_t` are signed types.

Red Zone

The ARM64 iOS red zone consists of the 128 bytes immediately below the stack pointer `sp`. As with the x86-64 ABI, the operating system has committed not to modify these bytes during exceptions. User-mode programs can rely on them not to change unexpectedly, and can potentially make use of the space for local variables.

In some circumstances, this approach can save an `sp`-update instruction on function entry and exit.

Note: If a function makes another call itself, then in general it must assume that the callee will modify the contents of its red zone and must therefore fall back to creating a proper stack frame.

Divergences from the Generic C++ ABI

The generic ARM64 C++ ABI is specified in [C++ Application Binary Interface Standard for the ARM 64-bit architecture](#), which is in turn based on the [Itanium C++ ABI](#) used by many UNIX-like systems.

Some sections are ELF-specific and not applicable to the underlying object format used by iOS. There are, however, some significant differences from these specifications in iOS.

Name Mangling

When compiling C++ code, types get incorporated into the names of functions in a process referred to as “mangling.” The iOS ABI differs from the generic specification in the following small ways.

- Because `va_list` is an alias for `char *`, it is mangled in the same way—as `Pc` instead of `St9__va_list`.
- NEON vector types are mangled in the same way as their 32-bit ARM counterparts, rather than using the 64-bit scheme. For example, iOS uses `17__simd128_int32_t` instead of the generic `11__Int32x4_t`.

Other Itanium Divergences

- In the generic ABI, empty structs are treated as aggregates with a single byte member for parameter passing. In iOS, however, they are ignored unless they have a nontrivial destructor or copy-constructor. If they do have such functions, they are considered as aggregates with one byte member in the generic manner.

- As with the ARM 32-bit C++ ABI, iOS requires the complete-object (C1) and base-object (C2) constructors to return `this` to their callers. Similarly, the complete object (D1) and base object (D2) destructors return `this`. This requirement is not made by the generic ARM64 C++ ABI.
- In the generic C++ ABI, array cookies change their size and alignment according to the type being allocated. As with the 32-bit ARM, iOS provides a fixed layout of two `size_t` words, with no extra alignment requirements.
- In iOS, object initialization guards are nominally `uint64_t` rather than `int64_t`. This affects the prototypes of the functions `__cxa_guard_acquire`, `__cxa_guard_release` and `__cxa_guard_abort`.
- In the generic ARM64 ABI, function pointers whose type differ only in being `extern "C"` or `extern "C++"` are interchangeable. This is not the case in iOS.

Data Types and Data Alignment

Using the correct data types for your variables helps to maximize the performance and portability of your programs. Data alignment specifies how data is laid out in memory. A data type's **natural alignment** specifies the default alignment of values of that type.

Table 1 lists the integer data types and their sizes and natural alignment in the ARM64 environment.

Table 1 Size and alignment of integer data types

Data type	Size (in bytes)	Natural alignment (in bytes)
<code>BOOL</code> , <code>bool</code>	1	1
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	8	8
<code>long long</code>	8	8
<code>pointer</code>	8	8
<code>size_t</code>	8	8
<code>NSInteger</code>	8	8
<code>CFIndex</code>	8	8

Data type	Size (in bytes)	Natural alignment (in bytes)
fpos_t	8	8
off_t	8	8

ARMv7 Function Calling Conventions

In general, applications built for the ARMv7 environment are capable of running in the ARMv6 environment and vice versa. This is because the calling conventions for the ARMv7 environment are nearly identical to those found in the ARMv6 environment. Therefore, this article describes only the places where the ARMv7 environment deviates or extends the ARMv6 environment.

For detailed information about the calling conventions for the ARMv6 environment, see [“ARMv6 Function Calling Conventions”](#) (page 12).

Function Calls

This section details the process of calling a subroutine and passing parameters to it, and how subroutines return values to their callers. The following sections highlight minor differences between the ARMv6 and ARMv7 environments.

Prologs and Epilogs

The version of Thumb available in ARMv7 is capable of saving and restoring the contents of the VFP registers. In addition, ARMv7 assembly uses a unified set of mnemonics; therefore, the same assembly code can be used to generate either ARM or Thumb machine code.

Listing 1 shows an example of a prolog that saves a number of key registers, including several VFP registers. It also allocates 36 bytes for local storage.

Listing 1 Example prolog for ARM or Thumb-2 (ARMv7)

```
push    {r4-r7, lr}           // save LR, R7, R4-R6
add     r7, sp, #12           // adjust R7 to point to saved R7
push    {r8, r10, r11}        // save remaining GPRs (R8, R10, R11)
vstmdb  sp!, {d8-d15}          // save VFP/Advanced SIMD registers D8
                                   // (aka S16-S31, Q4-Q7)
sub     sp, sp, #36           // allocate space for local storage
```

Listing 2 shows an example epilog that restores the registers saved by the preceding prolog.

Listing 2 Example epilog for ARM or Thumb-2 (ARMv7)

```
add    sp, sp, #36           // deallocate space for local storage
vldmia sp!, {d8-d15}         // restore VFP/Advanced SIMD registers
pop    {r8, r10, r11}        // restore R8-R11
pop    {r4-r7, pc}           // restore R4-R6, saved R7, return to saved LR
```

Register Preservation

Register preservation on the ARMv7 architecture is the same as that for the ARMv6 architecture, except for the changes and additions noted in Table 1.

Table 1 Additional processor registers in the ARMv7 architecture

Type	Name	Preserved	Notes
VFP register	D0-D7	No	Also known as Q0-Q3 on ARMv7. These registers are accessible from Thumb mode on ARMv7.
	D8-D15	Yes	Also known as Q4-Q7 on ARMv7. These registers are accessible from Thumb mode on ARMv7.
	D16-D31	No	Only available in ARMv7. Also known as Q8-Q15.

For information about all other registers, see [Table 2](#) (page 18).

ARMv6 Function Calling Conventions

When functions (routines) call other functions (subroutines), they may need to pass arguments to them. The called subroutines access these arguments as **parameters**. Conversely, some subroutines pass a **result** or return value to their callers. In the ARMv6 environment, arguments may be passed on the runtime stack or in registers; in addition, some vector arguments are also passed in registers. Results are returned in registers or in memory. To efficiently pass values between callers and callees, GCC follows strict rules when it generates a program's object code.

This article describes the data types that can be used to manipulate the arguments and results of subroutine calls, how routines pass arguments to the subroutines they call, and how subroutines that provide a return value pass the result to their callers. This article also lists the registers available in the ARMv6 architecture and whether their value is preserved after a subroutine call.

The function calling conventions used in the ARMv6 environment are the same as those used in the Procedure Call Standard for the ARM Architecture (release 1.07), with the following exceptions:

- The stack is 4-byte aligned at the point of function calls.
- Large data types (larger than 4 bytes) are 4-byte aligned.
- Register R7 is used as a frame pointer
- Register R9 has special usage.

The content of this article is largely based on the content in **Procedure Call Standard for the ARM Architecture (AAPCS)**, available at <http://infocenter.arm.com/help/topic/com.arm.doc.ih0042c/>

Data Types and Data Alignment

Using the correct data types for your variables helps to maximize the performance and portability of your programs. Data alignment specifies how data is laid out in memory. A data type's **natural alignment** specifies the default alignment of values of that type.

Table 1 lists the ANSI C scalar data types and their sizes and natural alignment in this environment.

Table 1 Size and natural alignment of the scalar data types

Data type	Size (in bytes)	Natural alignment (in bytes)
_Bool, bool	1	1
unsigned char	1	1
char, signed char	1	1
unsigned short	2	2
signed short	2	2
unsigned int	4	4
int, signed int	4	4
unsigned long	4	4
signed long	4	4
unsigned long long	8	4
signed long long	8	4
float	4	4
double	8	4
long double	8	4
pointer	4	4

These are some important details about this environment:

- A byte is 8 bits long.
- A null pointer has a value of 0.
- This environment uses the little-endian byte ordering scheme to store numeric and pointer data types. That is, the least significant bytes go first, followed by the most significant bytes.

These are the alignment rules followed in this environment:

1. Scalar data types use their natural alignment.

2. Composite data types (arrays, structures, and unions) take on the alignment of the member with the highest alignment. An array assumes the same alignment as its elements. The size of a composite data type is a multiple of its alignment (padding may be required).

Function Calls

The sections that follow detail the process of calling a subroutine and passing parameters to it, and how subroutines return values to their callers.

Note: These parameter-passing conventions are part of the Apple standard for procedural programming interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions.

Calling a Function

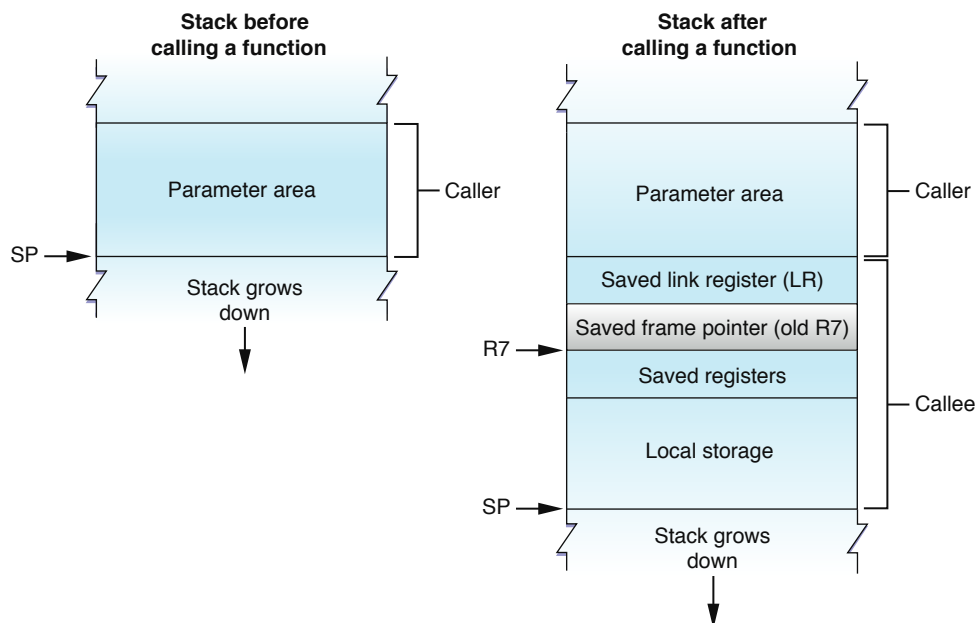
In iOS, you can call routines using either the Thumb or the ARM instruction sets. The main difference between these instruction sets is how you set up the stacks and parameter lists.

All subroutine call and return sequences must support interworking between ARM and Thumb states. This means that you must use the appropriate `BLX` and `BX` instructions (instead of the `MOV` instruction) for all calls to function pointers. For more information about using these instructions in your calls, see the AAPCS document.

Stack Structure

The ARM environment uses a stack that—at the point of function calls—is 4-byte aligned, grows downward, and contains local variables and a function’s parameters. Figure 1 shows the stack before and during a subroutine call. (To help prevent the execution of malicious code on the stack, clang protects the stack against execution.)

Figure 1 Stack layout



The **stack pointer** (SP) points to the bottom of the stack. Stack frames contain the following areas:

- **The parameter area** stores the arguments the caller passes to the called function or stores space for them, depending on the type of each argument and the availability of registers. This area resides in the caller’s stack frame.
- **The linkage area** contains the address of the caller’s next instruction.
- **The saved frame pointer** (optional) contains the base address of the caller’s stack frame.
- The **local storage area** contains the subroutine’s local variables and the values of the registers that must be restored before the called function returns. See [“Register Preservation”](#) (page 18) for details.
- The **saved registers area** contains the values of the registers that must be restored before the called function returns. See [“Register Preservation”](#) (page 18) for details.

In this environment, the stack frame size is not fixed.

Prologs and Epilogs

The called subroutine is responsible for allocating its own stack frame. This operation is accomplished by a section of code called the **prolog**, which the compiler places before the body of the function. After the body of the function, the compiler places an **epilog** to restore the process to the state it was prior to the subroutine call.

The prolog performs the following tasks:

1. Pushes the value of the link register (LR) onto the stack.
2. Pushes the value of the frame pointer (R7) onto the stack.
3. Sets the frame pointer (R7) to the value of the stack pointer (SP). (Updating R7 in this way gives the debugger a way to find previous stack frames.)
4. Pushes the values of the registers that must be preserved (see “[Register Preservation](#)” (page 18)) onto the stack.
5. Allocates space in the stack frame for local storage.

The epilog performs these tasks:

1. Deallocates the space used for local storage in the stack.
2. Restores the preserved registers (see “[Register Preservation](#)” (page 18)) by popping the values saved on the stack by the prolog.
3. Restores the value of the frame pointer (R7) by popping it from the stack.
4. Returns by popping the saved link register (LR) into the program counter (PC).

Note: Not all parts of the prolog and epilog are required. A routine that does not use a register does not need to save it. For example, if a routine does not use high registers (R8, R10, R11) or nonvolatile VFP registers, those registers do not need to be saved. In addition, leaf functions do not need to use the stack at all unless they need to save nonvolatile registers.

ARM Mode Examples

Listing 1 shows an example of a prolog in ARM mode. In this example, the prolog saves the contents of the VFP registers and allocates an additional 36 bytes of local storage.

Listing 1 Example prolog for ARM (ARMv6)

```
stmfd    sp!, {r4-r7, lr}    // save LR, R7, R4-R6
add      r7, sp, #12         // adjust R7 to point to saved R7
```



```
stmfd    sp!, {r8, r10, r11}    // save remaining GPRs (R8, R10, R11)
fstmfd   sp!, {d8-d15}          // save VFP registers D8-D15
                                   // (aka S16-S31 aka Q4-Q7)
sub       sp, sp, #36            // allocate space for local storage
```

Listing 2 shows an example of an epilog in ARM mode. This example deallocates the local storage and restores the registers saved in the prolog shown in [Listing 1](#) (page 16).

Listing 2 Example epilog for ARM (ARMv6)

```
add       sp, sp, #36            // deallocate space for local storage
fldmfd   sp!, {d8-d15}          // restore VFP registers
ldmfd    sp!, {r8, r10, r11}    // restore R8-R11
ldmfd    sp!, {r4-r7, pc}       // restore R4-R6, saved R7, return to saved LR
```

Thumb Mode Examples

Listing 3 shows an example of a prolog in Thumb. In this example, the prolog does not save nonvolatile VFP registers because Thumb-1 cannot access those registers.

Listing 3 Example prolog for Thumb (ARMv6)

```
push     {r4-r7, lr}            // save LR, R7, R4-R6
mov      r6, r11                 // move high registers to low registers so
mov      r5, r10                 // they can be saved. (Can be skipped if the
mov      r4, r8                  // routine does not use R8, R10 or R11)
push     {r4-r6}                 // save R8, R10, R11 (now in R4-R6)
add      r7, sp, #24             // adjust R7 to point to saved R7
sub      sp, #36                 // allocate space for local storage
```

Listing 4 shows a possible epilog in Thumb. This example restores the registers saved in the prolog shown in [Listing 3](#) (page 17).

Listing 4 Example epilog for Thumb (ARMv6)

```
add      sp, #36                 // deallocate space for local storage
pop      {r2-r4}                 // pop R8, R10, R11
mov      r8, r2                  // restore high registers
```

```
mov    r10, r3
mov    r11, r4
pop    {r4-r7, pc}           // restore R4-R6, saved R7, return to saved LR
```

Passing Arguments

The compiler generally adheres to the argument passing rules found in the AAPCS document. You should consult that document for the details of how arguments are passed but the following items are worth noting:

- In general, the first four scalar arguments go into the core registers (R0, R1, R2, R3) and any remaining arguments are passed on the stack. This may not always be the case though. For specific details on how arguments are mapped to registers or the stack, see the AAPCS document.
- Large data types (larger than 4 bytes) are 4-byte aligned.
- For floating-point arguments, the Base Standard variant of the Procedure Call Standard is used. In this variant, floating-point (and vector) arguments are passed in general purpose registers (GPRs) instead of in VFP registers)

Returning Results

The compiler generally adheres to the argument passing rules found in the AAPCS document. This means that most values are returned in R0 unless the size of the return value warrants different handling. For more information, see the AAPCS document.

Register Preservation

Table 2 lists the ARM architecture registers used in this environment and their volatility in procedure calls. Registers that must preserve their value after a function call are called **nonvolatile**.

Table 2 Processor registers in the ARMv6 architecture

Type	Name	Preserved	Notes
General-purpose register	R0-R3	No	These registers are used to pass arguments and results. They are available for general use within a routine and between function calls.
	R4-R6	Yes	
	R7	Yes	Frame pointer. Usually points to the previously saved stack frame and the saved link register.

Type	Name	Preserved	Notes
	R8	Yes	
	R9	Special	R9 has special restrictions that are described below.
	R10-R11	Yes	
	R12	No	R12 is the intra-procedure scratch register, also known as IP. It is used by the dynamic linker and is volatile across all function calls. However, it can be used as a scratch register between function calls.
	R13	Special	The stack pointer (SP).
	R14	Special	The link register (LR). Set to the return address on a function call.
	R15	Special	The program counter (PC).
Program status register	CPSR	Special	The condition bits (27-31) and GE bits (16-19) are not preserved by function calls. The E bit must remain zero (little-endian mode) when calling or returning from a function. The T bit should only be set by a branch routine. All other bits must not be modified.
VFP register	D0-D7	No	Also known as S0-S15. These registers are not accessible from Thumb mode on ARMv6.
	D8-D15	Yes	Also known as S16-S31. These registers are not accessible from Thumb mode on ARMv6.
VFP status register	FPSCR	Special	Condition code bits (28-31) and saturation bits (0-4) are not preserved by function calls. Exception control (8-12), rounding mode (22-23), and flush-to-zero (24) bits should be modified only by specific routines that affect the application state (including framework API functions). Short vector length (16-18) and stride (20-21) bits must be zero on function entry and exit. All other bits must not be modified.

The following notes should also be taken into consideration regarding register usage:

- Although the AAPCS document defines R7 as a general purpose nonvolatile register, iOS uses it as a frame pointer. Failure to use R7 as a frame pointer can prevent debugging and performance tools from generating valid backtraces. In addition, some ARM environments use the mnemonic FP to refer to R11. In iOS, R11 is a general-purpose nonvolatile register. As a result, the term FP is not used to avoid confusion.

- In iOS 2.x, register R9 is reserved for operating system use and must not be used by application code. Failure to do so can result in application crashes or aberrant behavior. However, in iOS 3.0 and later, register R9 can be used as a volatile scratch register. These guidelines differ from the general usage provided for by the AAPCS document.
- Accessing the VFP registers from Thumb mode (in ARMv6) is not supported. If you need access to the VFP registers, you must run your code in ARM mode. In iOS, switching between ARM and Thumb mode can be done only at function boundaries.

Document Revision History

This table describes the changes to *iOS ABI Function Call Guide*.

Date	Notes
2013-09-18	Added function calling conventions for ARM64.
2009-07-14	New document that describes the function-calling conventions used in iOS.



Apple Inc.
Copyright © 2013 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

SRS and the SRS Symbol are registered trademarks of SRS Labs, Inc.

UNIX is a registered trademark of The Open Group.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.