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 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)
`BOOL`, `bool`	1	1
`char`	1	1
`short`	2	2
`int`	4	4
`long`	8	8
`long long`	8	8
pointer	8	8
`size_t`	8	8
`NSInteger`	8	8
`CFIndex`	8	8
`fpos_t`	8	8

`off_t`	8	8