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 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.ihi0042c/

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 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

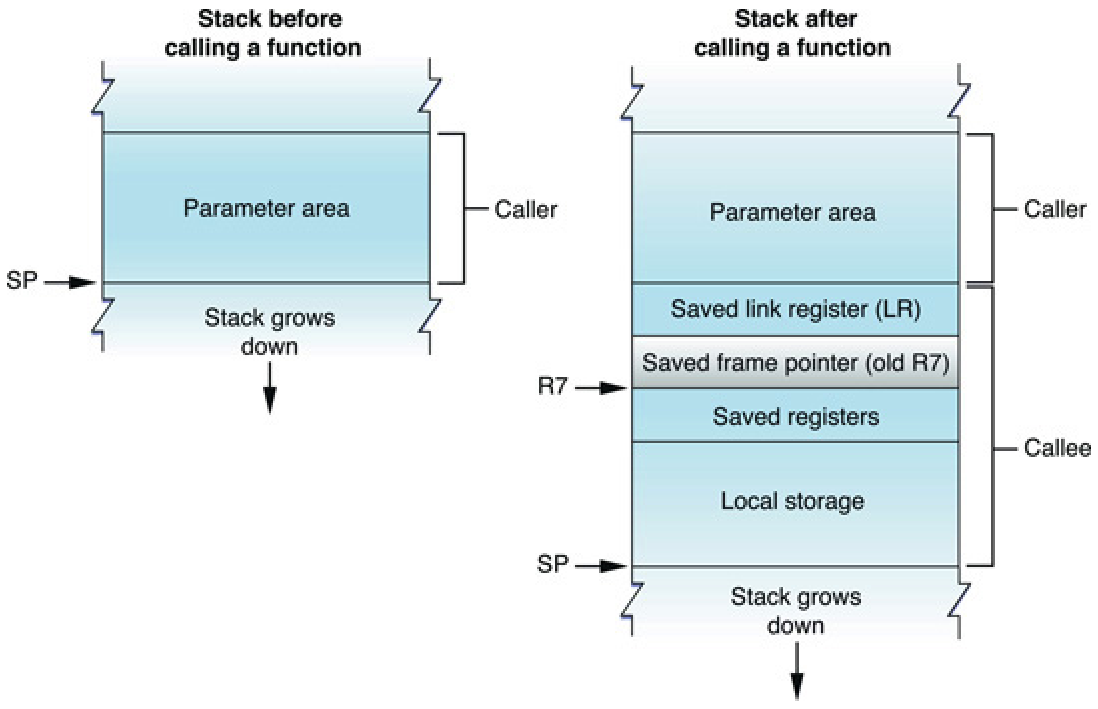
In iOS, you can call routines using either the Thumb or the ARM instruction sets. The main difference between these instruction sets is how you set up the stacks and parameter lists.

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 for details.
- The *saved registers area* contains the values of the registers that must be restored before the called function returns. See Register Preservation 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) 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) 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)
fstmfdd   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.

Listing 2 Example epilog for ARM (ARMv6)

```
add       sp, sp, #36            // deallocate space for local storage
fldmfdd   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.

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.
	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.