Mac OS X ABI Function Call Guide

General

2009-02-04

Contents

Figures, Tables, and Listings

Introduction to Mac OS X ABI Function Call Guide

This document describes the function-calling conventions used in the Mac OS X ABI on the architectures on which Mac OS X can run. Specifically, 32-bit PowerPC, 64-bit PowerPC, and IA-32.

The information in this document is based on Mac OS X v10.4 and later, and Xcode Tools 2.2 and later.

This document is intended for developers interested in the calling conventions used in the Mac OS X 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:

Each of these articles describes the data types that can be used to manipulate the arguments and results of function calls, how routines pass arguments to the functions they call, and how functions pass results to their callers. They also list the registers available in each architecture and whether their value is preserved after a function call.

See Also

The following documents contain information related to function calls in Mac OS X.

- *PowerPC Numerics* in Performance Documentation. Describes how floating-point operations are implemented in Mac OS X.

- *System V Application Binary Interface: Intel386 Architecture Processor Supplement*. Describes the data representation, register usage, stack management, and function-calling sequence the System V ABI uses in the IA-32 architecture. This document is located at http://www.caldera.com/developers/devspecs/abi386-4.pdf.

- *System V Application Binary Interface AMD64 Architecture Processor Supplement*, found at http://www.x86-64.org/documentation. Describes the System V x86-64 ABI, on which the function calling conventions used in the Mac OS X x86-64 environment are based.

32-bit PowerPC Function Calling Conventions

When functions (routines) call other functions (subroutines), they may need to pass arguments to the called functions. The called functions access those arguments as **parameters**. Conversely, some functions return a **result** or return value to their callers. Both arguments and results can be passed using the 32-bit PowerPC architecture registers or the runtime stack, depending on the data type of the values involved. For the successful and efficient passing of values between routines and subroutines, 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 function calls, how routines pass arguments to the subroutines they call, and how functions pass results to their callers. It also lists the registers available in the 32-bit PowerPC architecture and whether their value is preserved after a function call.

Data Types and Data Alignment

Using the correct data types for your variables and setting the appropriate **data alignment** for your data can maximize the performance and portability of your programs. Data alignment specifies how data is laid out in memory.

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 and natural alignment (in bytes)
`_Bool,bool`	4
`unsigned char`	1
`char,signed char`	1
`unsigned short`	2
`signed short`	2
`unsigned int`	4
`signed int`	4
`unsigned long`	4
`signed long`	4
`unsigned long long`	8

Data type	Size and natural alignment (in bytes)
`signed long long`	8
`float`	4
`double`	8
`long double`	16*
pointer	4

(*) In Mac OS X v10.4 and later and GCC 4.0 and later, the size of the `long double` extended precision data type is 16 bytes (it's made up of two 8-byte doubles). In earlier versions of Mac OS X and GCC, `long double` is equivalent to `double`. You should not use the `long double` type when you use GCC 4.0 or later to develop or in programs targeted at Mac OS X versions earlier than 10.4.

These are some important details about the 32-bit PowerPC environment:

- A byte is 8 bits long.

- A null pointer has a value of 0.

- This environment uses the big-endian byte ordering scheme to store numeric and pointer data types. That is, the most significant bytes go first, followed by the least significant bytes.

- This environment uses the two's-complement binary representation for signed integer data types.

- Arithmetic for the 64-bit integer data types must be synthesized by the compiler since the 32-bit PowerPC architecture does not implement 64-bit integer math operations.

- The `float` and `double` data types conform to the IEEE-754 standard representation. For the value range and precise format of floating-point data types, see *PowerPC Numerics* in Performance Documentation.

This environment supports multiple data alignment modes. The alignment of data types falls into two categories:

- **Natural alignment.** The alignment of a data type when allocated in memory or assigned a memory address.

  The natural alignment of a data type is its size. Table 1 (page 9) shows the natural alignment of each data type supported by this environment.

- **Embedding alignment.** The alignment of a data type within a composite data structure.

For example, the alignment of an `unsigned short` variable on the stack may differ from that of an `unsigned short` element embedded in a data structure.

The embedding alignment for data structures varies depending on the alignment mode selected. Generally, you can set the alignment mode using compiler options or `#pragma` statements. You should consider the compatibility and performance issues described later in this section when choosing a particular alignment mode.

These are the embedding alignment modes available in the 32-bit PowerPC environment:

- **Power alignment mode** is derived from the alignment rules used by the IBM XLC compiler for the AIX operating system. It is the default alignment mode for the PowerPC-architecture version of GCC used on AIX and Mac OS X. Because this mode is most likely to be compatible between PowerPC-architecture compilers from different vendors, it's typically used with data structures that are shared between different programs.

  The rules for power alignment are:

  - ❏ The embedding alignment of the first element in a data structure is equal to the element's natural alignment.

  - ❏ For subsequent elements with a natural alignment less than 4 bytes, the embedding alignment of each element is equal to its natural alignment.

  - ❏ For subsequent elements that have a natural alignment greater than 4 bytes, the embedding alignment is 4, unless the element is a `vector`.

  - ❏ The embedding alignment for `vector` elements is always 16 bytes.

  - ❏ The embedding alignment of a composite data type (array or data structure) is determined by the largest embedding alignment of its members.

  - ❏ The total size of a composite type is rounded up to a multiple of its embedding alignment, and is padded with null bytes.

  Because the natural alignment of the `double` and `long long` data types is greater than 4 bytes, they may not be appropriately aligned in power alignment mode. Any misalignment impairs performance when such data members are accessed. When you use these data types for any element after the first element, the compiler pads the structure to align the elements to the next multiple of their natural alignment.

- **Mac68K alignment mode** is usually used with legacy data structures inherited from Mac OS 9 and earlier systems. New code should not need to use this alignment mode except to preserve compatibility with older data structures.

  The rules for Mac68K alignment are:

  - ❏ The embedding alignment of the `char` data type is 1 byte.

  - ❏ The embedding alignment of all other data types (except `vector`) is 2 bytes.

  - ❏ The embedding alignment for the `vector` data type is 16 bytes.

  - ❏ The total size of a composite data type is rounded up to a multiple of 2 bytes.

- **Natural alignment mode** uses the natural alignment of each data type as its embedding alignment. Use this alignment mode to obtain the highest performance when using the `double`, `long long`, and `long double` data types.

- **Packed alignment mode** contains no alignment padding between elements (the alignment for all data types is 1 byte). Use this alignment mode when you need a data structure to use as little memory as possible. Note, however, that packed alignment can significantly lower the performance of your application.

> **Note:** Data items passed as parameters in a function call have special alignment rules. See "Stack Structure" (page 13) for more information.

Table 2 lists the alignment for structure fields of the fundamental data types and composite data types in the supported alignment modes.

Table 2   Alignment for structure fields

Data type	Power alignment	Natural alignment	Mac68K alignment	Packed alignment
`_Bool,bool`	4	4	2	1
`char`	1	1	1	1
`short`	2	2	2	1
`int`	4	4	2	1
`long`	4	4	2	1
`long long`	4 or 8	8	2	1
`float`	4	4	2	1
`double`	4 or 8	8	2	1
`long double`			2	1
`vector`	16	16	16	1
Composite (data structure or array)	4, 8, or 16	1, 2, 4, 8, or 16	2	1

With GCC you can control data-structure alignment by adding `#pragma` statements to your source code or by using command-line options. The power alignment mode is used if you do not specify otherwise.

To set the alignment mode, use the `gcc` flags `-malign-power`, `-malign-mac68k`, and `-malign-natural`. To use a specific alignment mode in a data structure, add this statement just before the data-structure declaration:

```
#pragma option align=<mode>
```

Replace `<mode>` with `power`, `mac68k`, `natural`, or `packed`. To restore the previous alignment mode, use `reset` as the alignment mode in a `#pragma` statement:

```
#pragma option align=reset
```

Function Calls

This section details the process of calling a function and passing arguments to it, and how functions return values to their callers.

> **Note:** These argument-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.

Stack Structure

This environment uses a stack that grows downward and contains linkage information, local variables, and a function's parameter information, as shown in Figure 1. (To help prevent the execution of malicious code on the stack, GCC protects the stack against execution.)

Figure 1   Stack layout
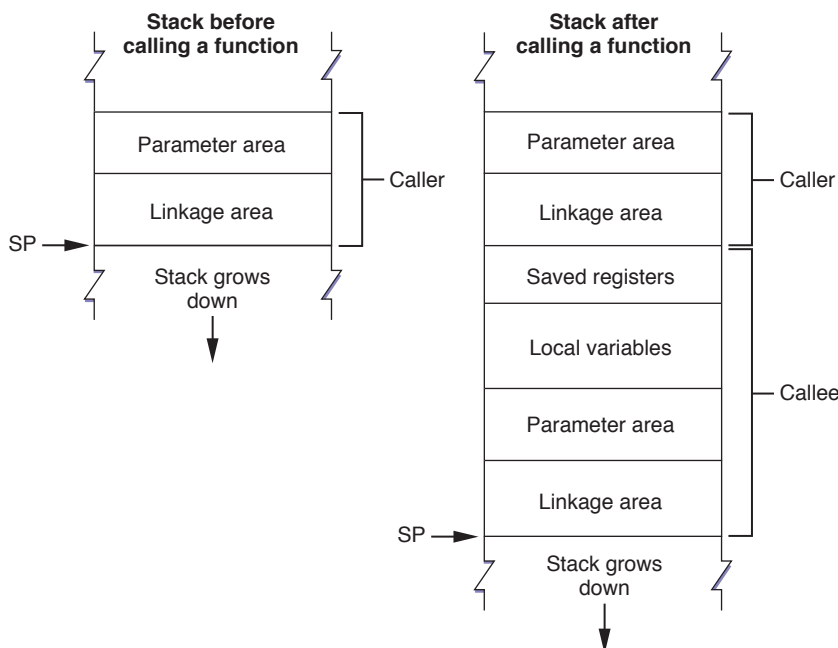
The **stack pointer** (SP) points to the bottom of the stack. The stack has a fixed frame size, which is known at compile time.

The calling routine's stack frame includes a **parameter area** and some linkage information. The parameter area has the arguments the caller passes to the called function or space for them, depending on the type of each argument and the availability of registers (see "Passing Arguments" (page 18) for details). Since the calling routine may call several functions, in the 32-bit PowerPC environment the parameter area is normally large enough to accommodate the largest argument list of all the functions the caller calls. It is the calling routine's responsibility to set up the parameter area before each function call. The called function is responsible for accessing the arguments placed in the parameter area.

The first 32 bytes in the parameter area correspond to the general-purpose registers GPR3 through GPR10. When data is placed in a general-purpose register and not duplicated in the parameter area, the corresponding section in the parameter area is reserved in case the called function needs to copy the value in the register to the stack. Table 3 shows the correspondence of parameter area locations to the general-purpose registers that can be used to pass arguments.

Table 3   Parameter area to general-purpose register mapping

Stack frame location	Register
SP+24	GPR3
SP+28	GPR4
SP+32	GPR5
SP+36	GPR6
SP+40	GPR7
SP+44	GPR8
SP+48	GPR9
SP+52	GPR10

These are the alignment rules followed when parameters are placed in the parameter area or in GPR3 through GPR10:

1. All nonvector parameters are aligned on 4-byte boundaries.

2. Vector parameters are aligned on 16-byte boundaries.

3. Noncomposite parameters (that is, parameters that are not arrays or data structures) smaller than 4 bytes occupy the high-order bytes of their 4-byte area.

4. Composite parameters (arrays, structures, and unions) 1 or 2 bytes in size occupy the low-order bytes of their 4-byte area. They are preceded by padding to 4 bytes.

    This rule is inconsistent with other 32-bit PowerPC binary interfaces. In AIX and Mac OS 9 (and earlier), padding bytes always follow the data structure even in the case of composite parameters smaller than 4 bytes.

5. Composite parameters 3 bytes or larger in size occupy the high-order bytes of their 4-byte area. They are followed by padding to make a multiple of 4 bytes, with the padding bytes being undefined.

For example, consider the `foo` function, declared like this:

```
void foo(SInt32 i1, float  f1, double d1, SInt16 s1, double d2,
         UInt8  c1, UInt16 s2, float  f2, SInt32 i2);
```

Table 4 shows how the function's arguments are assigned locations in the parameter area. The assignment takes into account the 4-byte alignment required for each argument.

Table 4   Parameter area layout for the `foo` call

Parameter	Type	Location	Data size and padding (in bytes)
i1	SInt32	SP+24	4, 0
f1	float	SP+28	4, 0
d1	double	SP+32	8, 0
s1	SInt16	SP+40	2, 2
d2	double	SP+44	8, 0
c1	UInt8	SP+52	1, 3
s2	UInt16	SP+56	2, 2
f2	float	SP+60	4, 0
i2	SInt32	SP+64	4, 0

The calling routine's **linkage area** holds a number of values, some of which are saved by the calling routine and some by the called function. The elements within the linkage area are:

- **The link register (LR).** Its value is saved at `8(SP)` by the called function if it chooses to do so. The link register holds the return address of the instruction that follows a branch and link instruction.

- **The condition register (CR).** Its value may be saved at `4(SP)` by the called function. The condition register holds the results of comparison operations. As with the link register, the called procedure is not required to save this value.

- **The stack pointer (SP).** Its value may be saved at `0(SP)` by the called function as part of its stack frame. **Leaf functions** are not required to save the the stack pointer. A leaf function is a function that does not call any other functions.

> **Note:** The space in the linkage area from `12(SP)` to `23(SP)` is reserved.

The linkage area is at the top of the stack, adjacent to the stack pointer. This positioning is necessary so that the calling routine can find and restore the values stored there and also allow the called function to find the caller's parameter area. This placement means that a routine cannot push and pop parameters from the stack once the stack frame is set up.

The stack frame also includes space for the called function's local variables. However, some registers are also available for use by the called function; see "Register Preservation" (page 22) for details. If the subroutine contains more local variables than would fit in the registers, it uses additional space on the stack. The size of the local-variable area is determined at compile time. Once a stack frame is allocated, the size of the local-variable area does not change.

Prologs and Epilogs

The called function is responsible for allocating its own stack frame, making sure to preserve 16-byte alignment in the stack. This operation is accomplished by a section of code called the **prolog**, which the compiler places before the body of the subroutine. After the body of the subroutine, the compiler places an **epilog** to restore the processor to the state it was prior to the subroutine call.

The compiler-generated prolog code does the following:

1. Decrements the stack pointer to account for the new stack frame and writes the previous value of the stack pointer to its own linkage area, which ensures the stack can be restored to its original state after returning from the call.

   It is important that the decrement and update tasks happen atomically (for example, with `stwu`, `stwux`, `stdu`, or `stdux`) so that the stack pointer and back-link are in a consistent state. Otherwise, asynchronous signals or interrupts could corrupt the stack.

2. Saves all nonvolatile general-purpose and floating-point registers into the saved-registers area. Note that if the called function does not change a particular nonvolatile register, it does not save it.

3. Saves the link-register and condition-register values in the caller's linkage area, if needed.

Listing 1 shows an example of a subroutine prolog. Notice that the order of these actions differs from the order previously described.

Listing 1   Example prolog

```
linkageArea = 24                                         ; size in 32-bit PowerPC ABI
params = 32                                              ; callee parameter area
localVars = 0                                            ; callee local variables
numGPRs = 0                                              ; volatile GPRs used by callee
numFPRs = 0                                              ; volatile FPRs used by callee

spaceToSave = linkageArea + params + localVars + 4*numGPRs + 8*numFPRs
spaceToSaveAligned = ((spaceToSave+15) & (-16))          ; 16-byte-aligned stack

_functionName:                                           ; PROLOG
    mflr        r0                                       ; extract return address
    stw         r0, 8(SP)                                ; save the return address
    stwu        SP, -spaceToSaveAligned(SP)              ; skip over caller save area
```

At the end of the subroutine, the compiler-generated epilog does the following:

1. Restores the nonvolatile general-purpose and floating-point registers that were saved in the stack frame.

   Nonvolatile registers are saved in the new stack frame before the stack pointer is updated only when they fit within the space beneath the stack pointer, where a new stack frame would normally be allocated, also known as the **red zone**. The red zone is by definition large enough to hold all nonvolatile general-purpose and floating-point registers but not the nonvolatile vector registers. See "The Red Zone" (page 17) for details.

2. Restores the condition-register and link-register values that were stored in the linkage area.

3. Restores the stack pointer to its previous value.

4. Returns control to the the calling routine using the address stored in the link register.

Listing 2 shows an example epilog.

Listing 2   Example epilog

```
                                        ; EPILOG
lwz         r0, spaceToSaveAligned + 8(SP)  ; get the return address
mtlr        r0                              ;    into the link register
addi        SP, SP, spaceToSaveAligned      ; restore stack pointer
blr                                         ;    and branch to the return address
```

The VRSAVE register is used to specify which vector registers must be saved during a thread or process context switch.Listing 3 shows an example prolog that sets up VRSAVE so that vector registers V0 through V2 are saved. Listing 3 also includes the epilog that restores VRSAVE to its previous state.

Listing 3   Example usage of the VRSAVE register

```
#define VRSAVE 256                          //  VRSAVE IS SPR# 256

    _functionName:
        mfspr   r2, VRSAVE                  ; get vector of live VRs
        oris        r0, r2, 0xE000          ; set bits 0-2 since we use V0..V2
        mtspr   VRSAVE, r0                  ; update live VR vector before using any
VRs

        ; Now, V0..V2 can be safely used.
        ; Function body goes here.

        mtspr   VRSAVE, r2                  ; restore VRSAVE
        blr                                 ; return to caller
```

The Red Zone

The space beneath the stack pointer, where a new stack frame would normally be allocated by a subroutine, is called the **red zone**. The red zone, shown in Figure 2, is considered part of the current stack frame. This area is not modified by asynchronous pushes, such as signals or interrupt handlers. Therefore, the red zone may be used for any purpose as long as a new stack frame does not need to be added to the stack. However, the contents of the red zone are assumed to be destroyed by any synchronous call.

Figure 2   The red zone
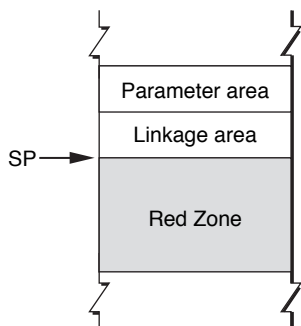
For example, because a leaf function does not call any other functions—and, therefore, does not allocate a parameter area on the stack—it can use the red zone. Furthermore, such a function does not need to use the stack to store local variables; it needs to save only the nonvolatile registers it uses for local variables. Since, by definition, no more than one leaf function is active at any time within a thread, there is no possibility of multiple leaf functions competing for the same red zone space.

A leaf function may or may not allocate a stack frame and decrement the stack pointer. When it doesn't allocate a stack frame, a leaf function stores the link register and condition register values in the linkage area of the routine that calls it (if necessary) and stores the values of any nonvolatile registers it uses in the red zone. This streamlining means that a leaf function's prolog and epilog do minimal work; they do not have to set up and take down a stack frame.

The size of the red zone is 224 bytes, which is enough space to store the values of nineteen 32-bit general-purpose registers and eighteen 64-bit floating-point registers, rounded up to the nearest 16-byte boundary. If a leaf function's red zone usage would exceed the red zone size, it must set up a stack frame, just as functions that call other functions do.

Passing Arguments

In the C language, functions can declare their parameters using one of three conventions:

■ The types of all parameters is specified in the function's prototype. For example:

```
int foo(int, short);
```

In this case, the type of all the function's parameters is known at compile time.

■ The function's prototype declares some fixed parameters and some nonfixed parameters. The group of nonfixed parameters is also called a **variable argument list**. For example:

```
int foo(int, ...);
```

In this case, the type of one of the function's parameters in known at compile time. The type of the nonfixed parameters is not known.

■ The function has no prototype or uses a pre–ANSI C declaration. For example:

```
int foo();
```

In this case, the type of all the function's parameters is unknown at compile time.

When the compiler generates the prolog for a function call, it uses the information from the function's declaration to decide how arguments are passed to the function. When the compiler knows the type of a parameter, it passes it in the most efficient way possible. But when the type is unknown, it passes the parameter using the safest approach, which may involve placing data both in registers and in the parameter area. For called functions to access their parameters correctly, it's important that they know when parameters are passed in the stack or in registers.

Arguments are passed in the stack, in registers, or both, depending on their types and the availability of registers. There are three types of registers: general purpose, floating point, and vector. General-purpose registers (GPRs) are 32-bit registers that can manipulate integral values and pointers. Floating-point registers (FPRs) are 64-bit registers that can manipulate single-precision and double-precision floating-point values. Vector registers are 128-bit registers that can manipulate 4 through 16 chunks of data in parallel.

The registers that can be used to pass arguments to called functions are the general-purpose registers GPR3 through GPR10, the floating-point registers FPR1 through FPR13, and the vector registers V2 through V13 (see "Register Preservation" (page 22) for details). These registers are also known as **parameter registers**.

> **Important:** Only the low 32 bits in each of the general-purpose registers available on the 64-bit PowerPC architecture are used in this environment. That is, only the low 32 bits of nonvolatile registers are saved and restored. However, all 64 bits are saved across asynchronous events, such as signals and preemptions. Therefore, you can use the 64 bits in each register between function calls. You control this feature through the `gcc` options `-arch` and `-mcpu`.

Typically, the called routine obtains arguments from registers. However, the caller generates a parameter area in the caller's stack frame that is large enough to hold all the arguments passed to the called function, regardless of how many of the arguments are actually passed in registers. (You can think of the parameter area as a data structure that has space to hold all the arguments in a given call.) There are several reasons for these scheme:

■ It provides the called function with space in the stack to store a register-based parameter if it wants to use one of the parameter registers for some other purpose. For example, the callee can use these space to pass arguments to a function it calls.

■ Functions with variable argument lists must often access their parameters from RAM, not from registers. Such functions must reserve 32 bytes (8 registers) in the parameter area to hold the parameter values.

To simplify debugging, GCC writes parameters from the parameter registers into the parameter area in the stack frame. This allows you to see all the parameters by looking only at the parameter area.

The compiler uses the following rules when passing arguments to subroutines:

■ Parameters whose type is known at compile time are processed as follows:

1. Scalar, non–floating-point elements are placed in the general-purpose registers GPR3 through GPR10. As each register is used, the caller allocates the register's corresponding section in the parameter area, as described in "Stack Structure" (page 13). When general-purpose registers are exhausted, the caller places scalar, non–floating-point elements in the parameter area.

2. The caller places floating-point parameters in the floating-point registers FPR1 through FPR13. As each floating-point register is used, the caller skips one or more general-purpose registers, based on the size of the parameter. (For example, a `float` element causes one (4-byte) general-purpose register to be skipped. A `double` element causes two general-purpose registers to be skipped.) When floating-point registers are exhausted, the caller places floating-point elements in the parameter area.

3. The caller places structures (`struct` elements) with only one noncomposite member in general-purpose or floating-point registers, depending on whether the member is an integer or a floating-point value. For example, the caller places a structure comprised of a `float` member in a floating-point register, not a general-purpose register. When registers of the required type are exhausted, the caller places structures in the parameter area.

4. The caller places `vector` parameters in vector registers V2 through V13. For procedures with a fixed number of parameters, the presence of vectors doesn't affect the allocation of general-purpose registers and floating-point registers. The caller doesn't allocate space for `vector` elements in the parameter area of its stack frame unless the number of `vector` elements exceeds the number of usable vector registers.

5. When the number of parameters exceeds the number of usable registers, the caller places the excess parameters in the parameter area.

■ Parameters whose type is not known at compile time (functions with variable-argument lists or using pre–ANSI C prototypes) are processed as follows:

1. The caller places nonvector elements both in general-purpose registers and in floating-point registers.

    Because the compiler doesn't know the type of the parameter, it cannot determine whether the argument should be passed in a general-purpose register or in a floating-point register. Therefore, callers place each argument in a floating-point register *and* the corresponding general-purpose registers based on the argument's size.

2. The caller places `vector` elements in vector registers *and* general-purpose registers (each vector element requires four general-purpose registers. The caller also allocates space in the parameter area that corresponds to the general-purpose registers used.

> **Important:** When the return type of the called function is a composite value (for example, `struct` or `union`), the caller passes a pointer in GPR3 as an implicit first parameter of the called function. Therefore, the functions' declared parameters start at GPR4. The pointer points to a section of memory large enough to hold the return value. See "Returning Results" (page 22) for more information.

For example, consider the `foo` function, declared like this:

```
void foo(SInt32 i1, float  f1, double d1, SInt16 s1, double d2,
         UInt8  c1, UInt16 s2, float  f2, SInt32 i2);
```

The caller places each argument to `foo` in a general-purpose register, a floating-point register, or the parameter area, depending on the parameter's data type and register availability. Table 5 describes this process.

Table 5   Assigning parameters to registers and the parameter area

Parameter	Type	Placed in	Reason
`i1`	`SInt32`	GPR3	Noncomposite, non–floating-point element.
`f1`	`float`	FPR1	Floating-point element. GPR4 is skipped.
`d1`	`double`	FPR2	Double-precision, floating-point element. GPR5 and GPR6 are skipped.
`s1`	`SInt16`	GPR7	Noncomposite, non–floating-point element.
`d2`	`double`	FPR3	Double-precision, floating-point element. GPR8 and GPR9 are skipped.
`c1`	`UInt8`	GPR10	Noncomposite, non–floating-point element.
`s2`	`UInt16`	SP+56, low half of word	No general-purpose registers available.
`f2`	`float`	FPR4	Floating-point element.

Parameter	Type	Placed in	Reason
i2	SInt32	SP+60	No general-purpose registers available.

> **Note:** In this case, the caller doesn't place any arguments that it places in general-purpose registers or floating-point registers in the parameter area.

Figure 3 illustrates the assignment of the `foo` parameters to registers and the parameter area. Keep in mind that the only parameters placed in the parameter area are `s2` and `i2`.

Figure 3   Assignment of parameters to registers and the parameter area
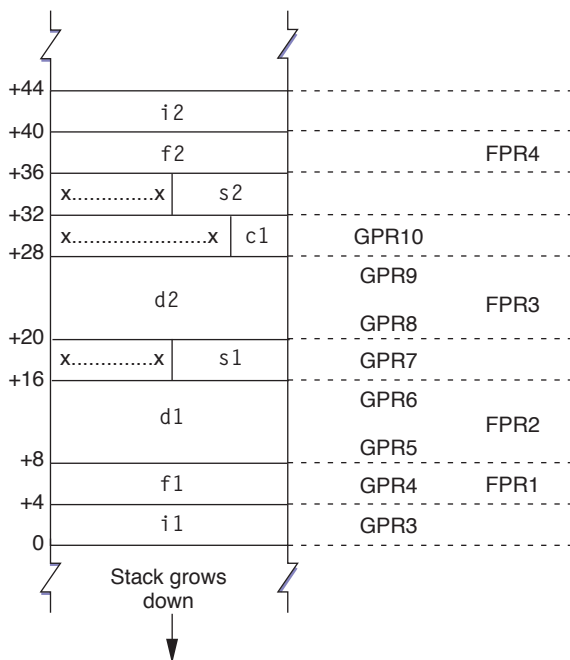
The called function can access the fixed parameters as usual. But it copies the general-purpose registers to the parameter area and accesses the values from there. Listing 4 shows a routine that accesses undefined parameters by walking through the stack.

Listing 4   A variable-argument procedure

```
#include <stdarg.h>
double dsum(int count, ...) {
    double sum = 0.0;
    double val;
    va_list arg;
    va_start(arg, count);
    while (count > 0) {
        val = va_arg(arg, double);
        sum += val;
        count--;
    }
    va_end(arg);
```

```
    return sum;
}
```

Returning Results

The following list describes where a function's return value is passed to the caller.

■ Scalars smaller than 4 bytes (such as `char` and `short`) are placed in the low word of GPR3. The register's high word is undefined.

■ Scalars 4 bytes in size (such as `long`, `int`, and pointers, including array pointers) are placed in GPR3.

■ Values of type `long long` are returned in the high word of GPR3 and the low word of GPR4.

■ Floating-point values are placed in FPR1.

■ Composite values (such as `struct` and `union`) and values larger than 4 bytes are placed at the location pointed to by GPR3. See "Passing Arguments" (page 18) for more information.

Register Preservation

Table 6 lists the 32-bit PowerPC architecture registers used in this environment and their volatility in function calls. Registers that must preserve their value after a function call are called **nonvolatile**.

Table 6   Processor registers in the 32-bit PowerPC architecture

Type	Name	Preserved	Notes
General-purpose register	GPR0	No	
	GPR1	Yes	Used as the stack pointer to store parameters and other temporary data items.
	GPR2	No	Available for general use.
	GPR3	No	The caller passes parameter values to the called procedure in GPR3 through GPR10. The caller may also pass the address to storage where the callee places its return value in this register.
	GPR4–GPR10	No	Used by callers to pass parameter values to called functions (see notes for GPR3).
	GPR11	Yes in nested functions. No in leaf functions.	In nested functions, the caller passes its stack frame to the nested function in this register. In leaf functions, the register is available. For details on nested functions, see the GCC documentation. This register is also used by lazy stubs in dynamic code generation to point to the lazy pointer.

Type	Name	Preserved	Notes
	GPR12	No	Set to the address of the branch target before an indirect call for dynamic code generation. This register is not set for a function that has been called directly; therefore, functions that may be called directly should not depend on this register being set up correctly. See *Mach-O Programming Topics* for more information.
	GPR13–GPR31	Yes	
Floating-point register	FPR0	No	
	FPR1–FPR13	No	Used to pass floating-point parameters in function calls.
	FPR14–FPR31	Yes	
Vector register	V0–V19	No	The caller passes vector parameters in V2 to V13 during a function call.
	V20–V31	Yes	
Special-purpose vector register	VRSAVE	Yes	32-bit special-purpose register. Each bit in this register indicates whether the corresponding vector register must be saved during a thread or process context switch.
Link register	LR	No	Stores the return address of the calling routine that called the current subroutine.
Count register	CTR	No	
Fixed-point exception register	XER	No	
Condition register fields	CR0, CR1	No	
	CR2–CR4	Yes	
	CR5–CR7	No	

64-bit PowerPC Function Calling Conventions

When functions (routines) call other functions (subroutines), they may need to pass arguments to them. These subroutines access those arguments as **parameters**. Conversely, some functions pass a **result** or return value to their callers. Both arguments and results can be passed using the 64-bit PowerPC architecture registers or the runtime stack, depending on the data type of the values involved. For the successful and efficient passing of values between routines and subroutines, 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 function calls, how routines pass arguments to the subroutines they call, and how functions pass results to their callers. It also lists the registers available in the 64-bit PowerPC architecture and whether their value is preserved after a function call.

Data Types and Data Alignment

Using the correct data types for your variables and setting the appropriate **data alignment** for your data can maximize the performance and portability of your programs. Data alignment specifies how data is laid out in memory.

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 and natural alignment (in bytes)
_Bool, bool	1
unsigned char	1
char, signed char	1
unsigned short	2
signed short	2
unsigned int	4
signed int	4
unsigned long	8
signed long	8
unsigned long long	8

Data type	Size and natural alignment (in bytes)
`signed long long`	8
`float`	4
`double`	8
`long double`	16
pointer	8

These are some important details about the 64-bit PowerPC environment:

- A byte is 8 bits long.

- A null pointer has a value of 0.

- This environment uses the big-endian byte ordering scheme to store numeric and pointer data types. That is, the most significant bytes go first, followed by the least significant bytes.

- This environment uses the two's-complement binary representation for signed integer data types.

- The `float` and `double` data types conform to the IEEE-754 standard representation. For the value range and precise format of floating-point data types, see *PowerPC Numerics* in Performance Documentation.

This environment supports multiple data alignment modes. Alignment of data types falls into two categories:

- **Natural alignment.** The alignment of a data type when allocated in memory or assigned a memory address.

  The natural alignment of a data type is its size. Table 1 (page 25) shows the natural alignment of each data type supported by this environment.

- **Embedding alignment.** The alignment of a data type within a composite data structure.

For example, the alignment of an `unsigned short` variable on the stack may differ from that of an `unsigned short` data item embedded in a data structure.

The embedding alignment for data structures varies depending on the alignment mode selected. Generally, you can set the alignment mode using compiler options or `#pragma` statements. You should consider the compatibility and performance issues described later in this section when choosing a particular alignment mode.

These are the embedding alignment modes available in the 64-bit PowerPC environment:

- **Power alignment mode** is derived from the alignment rules used by the IBM XLC compiler for the AIX operating system. It is the default alignment mode for the PowerPC-architecture version of GCC used on AIX and Mac OS X. Because this mode is most likely to be compatible between PowerPC-architecture compilers from different vendors, it's typically used with data structures that are shared between different programs.

  The rules for power alignment are:

  - ❏ The embedding alignment of the first element in a data structure is equal to the element's natural alignment.

❏ For subsequent elements with a natural alignment less than 4 bytes, the embedding alignment of each element is equal to its natural alignment.

❏ For subsequent elements that have a natural alignment greater than 4 bytes, the embedding alignment is 4, unless the element is a `vector`.

❏ The embedding alignment for `vector` elements is always 16 bytes.

❏ The embedding alignment of a composite data type (array or data structure) is determined by the largest embedding alignment of its members.

❏ The total size of a composite type is rounded up to a multiple of its embedding alignment, and is padded with null bytes.

Because the natural alignment of `double` and `long long` data types is greater than 4 bytes, they may not be appropriately aligned in power-alignment mode. Any misalignment impairs performance when such data members are accessed. When you use these data types for any element after the first element, the compiler pads the structure to align the elements to their natural alignment.

■ **Natural alignment mode** uses the natural alignment of each data type as its embedding alignment. Use this alignment mode to obtain the highest performance when using `double`, `long`, `long long`, and `long double` data types.

■ **Packed alignment mode** contains no alignment padding between elements (the alignment for all data types is 1 byte). Use this alignment mode when you need a data structure to use as little memory as possible. Note, however, that packed alignment can significantly lower the performance of your application.

> **Note:** Data items passed as parameters in a subroutine call have special alignment rules. See "Stack Structure" (page 28) for more information.

Table 2 lists the alignment for structure fields of the fundamental data types and composite data types in the supported alignment modes.

Table 2   Alignment for structure fields

Data type	Natural alignment	Power alignment	Packed alignment
`_Bool,bool`	1	1	1
`char`	1	1	1
`short`	2	2	1
`int`	4	4	1
`long`	8	4	1
`long long`	8	4	1
`float`	4	4	1
`double`	8	4 or 8	1
`long double`	8	8	1

Data type	Natural alignment	Power alignment	Packed alignment
`vector`	16	16	1
Composite (data structure or array)	1, 2, 4, 8, or 16	4, 8, or 16	1

With GCC you can control data-structure alignment by adding `#pragma` statements to your source code or by using command-line options. The power alignment mode is used if you do not specify otherwise.

To set the alignment mode, use the `gcc` flags `-malign-power` and `-malign-natural`. To use a specific alignment mode in a data structure, add this statement just before the data-structure declaration:

```
#pragma option align=<mode>
```

Replace `<mode>` with `power`, `natural`, or `packed`. To restore the previous alignment mode, use `reset` as the alignment mode in a `#pragma` statement:

```
#pragma option align=reset
```

Function Calls

This section details the process of calling a subroutine and passing arguments to it, and how functions 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.

Stack Structure

This environment uses a stack that grows downward and contains linkage information, local variables, and a subroutine's parameter information, as shown in Figure 1. (To help prevent the execution of malicious code on the stack, GCC protects the stack against execution.)

Figure 1   Stack layout
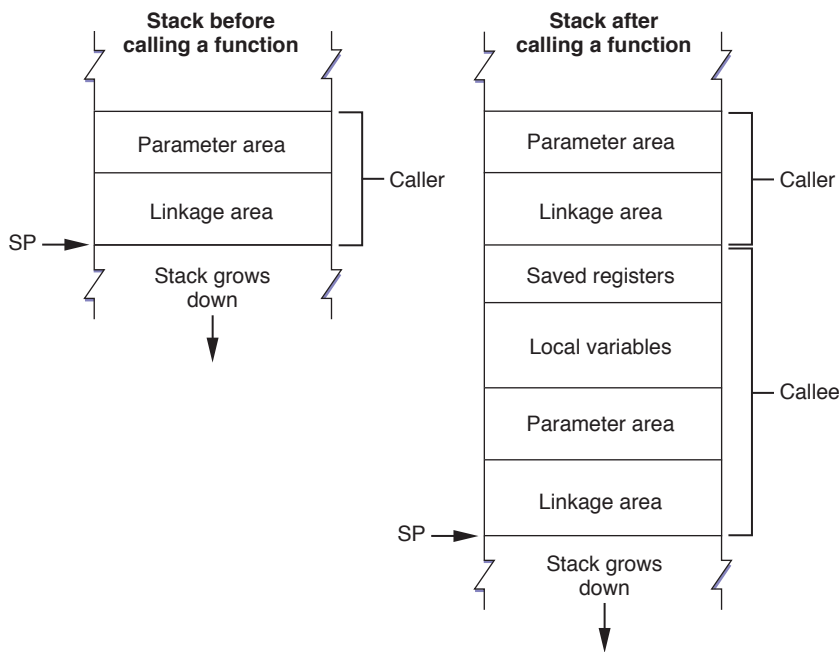
The **stack pointer** (SP) points to the bottom of the stack. The stack has a fixed frame size, which is known at compile time.

The calling routine's stack frame includes a **parameter area** and some linkage information. The parameter area has the arguments the caller passes to the called subroutine or space for them, depending on the type of each parameter and the availability of registers (see "Passing Arguments" (page 34) for details). Since the calling routine may call several subroutines, the parameter area must be large enough to accommodate the largest argument list of all the subroutines the caller calls. It is the calling routine's responsibility to set up the parameter area before each function call. The called function is responsible for accessing the arguments placed in the parameter area.

Bytes 48 through 112 of the parameter area correspond to the general-purpose registers GPR3 through GPR10. When data is placed in a general-purpose register and not duplicated in the parameter area, the corresponding section in the parameter area is reserved in case the called subroutine needs to copy the value in the register to the stack. Table 3 shows the correspondence of parameter-area locations to the general-purpose registers that can be used to pass parameters.

Table 3   Parameter area to general-purpose register mapping

Stack frame location	Register
SP+48	GPR3
SP+56	GPR4
SP+64	GPR5
SP+72	GPR6
SP+80	GPR7

Stack frame location	Register
SP+88	GPR8
SP+96	GPR9
SP+104	GPR10

When space is allocated for a parameter in the parameter area, the space allocated may be larger than the parameter's type. In this case, the parameter is "promoted" to a larger data type. Each parameter's address is the address of the previous parameter plus the size of the previous parameter's promoted type.

These are the promotion and alignment rules followed when parameters are placed in the parameter area or in general-purpose registers:

1. Integers are promoted to `long`. For example, `short` elements are sign-extended to 64-bits, and `unsigned int` elements are zero-padded on the left to 64-bits.

2. Floating-point elements are promoted to `double`.

3. Composite arguments (arrays and structures) are processed this way:

   a. The aligned size is computed by adding necessary padding to make it a multiple of the alignment.

   b. If the aligned size is 1, 2 or 4, the argument is preceded by padding to 4 bytes.

   c. Otherwise, the argument is followed by padding to make its size a multiple of 4 bytes, with the padding bytes being undefined. (GCC pads with `0`.

4. Parameters with a 16-byte natural alignment (for example, vectors or structures containing a vector), are 16-byte aligned.

For example, assume the function `foo` is declared like this:

```
int foo(int i, float f, long l, vector int v,
        double d, void* p, char c, short s);
```

The layout of the parameter area would be as shown in Table 4.

Table 4   Parameter area layout for the `foo` call

Parameter	Declared type	Promoted type	Location
i	int	long	SP+48: **Start of the parameter area.**
f	float	double	SP+56: `56 = 48 + sizeof(long)`
l	long	long	SP+64: `64 = 56 + sizeof(double)`
v	vector	vector	SP+80: `80 = align16(64 + sizeof(long))`
d	double	double	SP+96: `96 = 80 + sizeof(vector)`

Parameter	Declared type	Promoted type	Location
p	`void*`	`void*`	`SP+104:104 = 96 + sizeof(double)`
c	`char`	`long`	`SP+112:112 = 104 + sizeof(void*)`
s	`short`	`long`	`SP+120:120 = 112 + sizeof(long)`

The calling routine's **linkage area** holds a number of values, some of which are saved by the calling routine and some by the called subroutine. The elements within the linkage area are:

■ **The link register (LR).** It's value is saved at `16(SP)` by the called function if it chooses to do so. The link register holds the return address of the instruction that follows a branch and link instruction.

■ **The condition register (CR).** It's value may be saved at `8(SP)` by the called function. The condition register holds the results of comparison operations. As with the link register, the called subroutine is not required to save this value. Because the condition register is a 32-bit register, bytes 12 through 15 of the stack frame are unused but reserved.

■ **The stack pointer (SP).** It's value may be saved at `0(SP)` by the called function as part of its stack frame. **Leaf subroutines** are not required to save the the stack pointer. A leaf function is a routine that does not call any other function.

> **Note:** The space in the linkage area from `24(SP)` to `47(SP)` is reserved.

The linkage area is at the top of the stack frame, adjacent to the stack pointer. This positioning is necessary so the calling routine can find and restore the values stored there and also allow the called subroutine to find the caller's parameter area. This placement means that a routine cannot push and pop parameters from the stack once the stack frame is set up.

The stack frame also includes space for the called function's local variables. However, some registers are available for use by the called function; see "Register Preservation" (page 40) for details. If the subroutine contains more local variables than would fit in the registers, it uses additional space on the stack. The size of the local-variable area is determined at compile time. Once a stack frame is allocated, the size of the local-variable area cannot change.

Prologs and Epilogs

The called function is responsible for allocating its own stack frame, making sure to preserve 16-byte alignment in the stack. This operation is accomplished by a section of code called the **prolog**, which the compiler places before the body of the subroutine. After the body of the subroutine, the compiler places an **epilog** to restore the processor to the state it was prior to the subroutine call.

The compiler-generated prolog code does the following:

1. Decrements the stack pointer to account for the new stack frame and writes the previous value of the stack pointer into its own linkage area, which ensures the stack can be restored to its original state after returning from the call.

It is important that the decrement and update tasks happen atomically (for example, with `stwu`, `stwux`, `stdu`, or `stdux`) so that the stack pointer and back-link are in a consistent state. Otherwise, asynchronous signals or interrupts could corrupt the stack.

2. Saves all nonvolatile general-purpose and floating-point registers into the saved-registers area. Note that if the called function does not change a particular nonvolatile register, it does not save it.

3. Saves the link-register and condition-register values in the caller's linkage area, if needed.

Listing 1 shows an example of a routine prolog. Notice that the order of these actions differs from the order previously described.

Listing 1   Example prolog

```
linkageArea = 48                                       ; size in 64-bit PowerPC ABI
params = 64                                            ; callee parameter area
localVars = 0                                          ; callee local variables
numGPRs = 0                                            ; volatile GPRs used by callee
numFPRs = 0                                            ; volatile FPRs used by callee

spaceToSave = linkageArea + params + localVars + 8*numGPRs + 8*numFPRs
spaceToSaveAligned = ((spaceToSave+15) & (-16))        ; 16-byte-aligned stack

_functionName:                                         ; PROLOG
    mflr        r0                                     ; extract return address
    std         r0, 16(SP)                             ; save the return address
    stdu        SP, -spaceToSaveAligned(SP)            ; skip over caller save area
```

At the end of the function, the compiler-generated epilog does the following:

1. Restores the nonvolatile general-purpose and floating-point registers that were saved in the stack frame.

   Nonvolatile registers are saved in the new stack frame before the stack pointer is updated only when they fit within the space beneath the stack pointer, where a new stack frame would normally be allocated, also known as the **red zone**. The red zone is by definition large enough to hold all nonvolatile general-purpose and floating-point registers but not the nonvolatile vector registers. See "The Red Zone" (page 33) for details.

2. Restores the condition-register and link-register values that were stored in the linkage area.

3. Restores the stack pointer to its previous value.

4. Returns control to the calling routine using the address stored in the link register.

Listing 2 shows an example epilog.

Listing 2   Example epilog

```
                                          ; EPILOG
ld          r0, spaceToSaveAligned + 16(SP)   ; get the return address
mtlr        r0                                ;    into the link register
addi        SP, SP, spaceToSaveAligned        ; restore stack pointer
blr                                           ; and branch to the return address
```

The VRSAVE register is used to specify which vector registers must be saved during a thread or process context switch.Listing 3 shows an example prolog that sets up VRSAVE so that vector registers V0 through V2 are saved. Listing 3 also includes the epilog that restores VRSAVE to its previous state.

Listing 3   Example usage of the VRSAVE register

```
#define VRSAVE 256                         //  VRSAVE IS SPR# 256

    _functionName:
        mfspr    r2, VRSAVE                ; get vector of live VRs
        oris         r0, r2, 0xE000        ; set bits 0-2 since we use V0..V2
        mtspr    VRSAVE, r0                ; update live VR vector before using any
VRs

        ; Now, V0..V2 can be safely used.
        ; Function body goes here.

        mtspr    VRSAVE, r2                ; restore VRSAVE
        blr                                ; return to caller
```

The Red Zone

The space beneath the stack pointer, where a new stack frame would normally be allocated by a subroutine, is called the **red zone**. The red zone, shown in Figure 2, is considered part of the topmost (current) stack frame. This area is not modified by asynchronous pushes, such as signals or interrupt handlers. Therefore, the red zone may be used for any purpose as long as a new stack frame does not need to be added to the stack. However, the contents of the red zone are assumed to be destroyed by any synchronous call.

Figure 2   The red zone
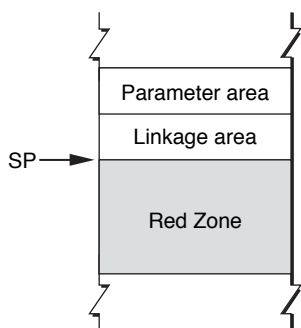
For example, because a leaf function does not call any other functions—and, therefore, does not allocate a parameter area on the stack—it can use the red zone. Furthermore, such a function does not need to use the stack to store local variables; it needs to save only the nonvolatile registers that it uses for local variables. Since by definition no more than one leaf function is active at any time within a thread, there is no possibility of multiple leaf functions competing for the same red zone space.

A leaf function may or may not allocate a stack frame and decrement the stack pointer. When it doesn't allocate a stack frame, a leaf function stores the link register and condition register values in the linkage area of the routine that calls it (if necessary) and stores the values of any nonvolatile registers it uses in the red zone. This streamlining means that a leaf function's prolog and epilog do minimal work; they do not have to set up and take down a stack frame.

The size of the red zone is 288 bytes, which is enough space to store the values of nineteen 64-bit general-purpose registers and eighteen 64-bit floating-point registers, rounded up to the nearest 16-byte boundary. If a leaf function's red zone usage would exceed the red zone size, it must set up a stack frame, just as functions that call other functions do.

Passing Arguments

In the C language, functions can declare their parameters using one of three conventions:

- The types of all parameters is specified in the function's prototype. For example:

  ```
  int foo(int, short);
  ```

  In this case, the type of all the function's parameters is known at compile time.

- The function's prototype declares some fixed parameters and some nonfixed parameters. The group of nonfixed parameters is also called a **variable argument list**. For example:

  ```
  int foo(int, ...);
  ```

  In this case, the type of one of the function's parameters in known at compile time. The type of the nonfixed parameters is not known.

- The function has no prototype or uses a pre–ANSI C declaration. For example:

  ```
  int foo();
  ```

  In this case, the type of all the function's parameters is unknown at compile time.

When the compiler generates the prolog for a function call, it uses the information from the function's declaration to decide how to pass arguments to the function. When the compiler knows the type of a parameter, it passes it in the most efficient way possible. But when the type is unknown, it passes the parameter using the safest approach, which may involve placing data both in registers and in the parameter area. For called functions to access their parameters correctly, it's important that they know when parameters are passed in the stack or in registers.

Arguments are passed in the stack or in registers depending on their types and the availability of registers. There are three types of registers: general purpose, floating point, and vector. General-purpose registers (GPRs) are 64-bit registers that can manipulate integral values and pointers. Floating-point registers (FPRs) are 64-bit registers that can manipulate single-precision and double-precision floating-point values. Vector registers are 128-bit registers that can manipulate 4 through 16 chunks of data in parallel.

The registers that can be used to pass arguments to called functions are the general-purpose registers GPR3 through GPR10, the floating-point registers FPR1 through FPR13, and the vector registers V2 through V13 (see "Register Preservation" (page 40) for details). These registers are also known as **parameter registers**.

The compiler uses the following rules when passing arguments to subroutines:

- Parameters whose promoted type is known at compile time are processed using these rules (see "Stack Structure" (page 28) for details on a parameter's promoted type):

  1. The caller places floating-point elements (except `long double` elements) in floating-point registers FPR1 through FPR13. As each floating-point register is used, the caller skips the next available general-purpose register. When floating-point registers are exhausted, the caller places these elements in the parameter area.

2. The caller places `long double` elements—which use a pair of `float` elements—in two floating-point registers. As each pair of floating-point registers is used, the caller skips the next two available general-purpose registers. When floating-point registers are exhausted, the caller places these elements in the parameter area.

3. The caller places `vector` elements in vector registers V2 through V13. Vector-register usage doesn't affect the availability of general-purpose registers. That is, no general-purpose registers are skipped as a result of using a vector register. When vector registers are exhausted, the caller places these elements in the parameter area.

4. The caller places elements of all other data types—including `complex` (defined in `complex.h`)—in general-purpose registers GPR3 through GPR10, when available. When general-purpose registers are exhausted, the caller places these elements in the parameter area.

   Structures that are 16 bytes in size are handled as if they were a pair of 64-bit integers. Therefore, they are placed in two general-purpose registers. Examples of structures that meet this criterion include a structure containing four `float` fields and a structure containing two `double` fields. Structures that contain three `float` fields, for example, are be processed using rule 5.

5. The caller recursively processes the members of structures passed by value and containing no unions:

   ❑ Floating-point fields are processed using rule 1 or rule 2, depending on their type.

   ❑ Vector fields are processed using rule 3.

   ❑ Fields of all other types—including arrays—are processed using rule 4.

■ Arguments to a pre–ANSI C–declared function are processed as follows:

1. The caller places floating-point elements in floating-point registers and general-purpose registers, when available. Otherwise, the caller places them in the parameter area.

2. The caller places `vector` elements in vector registers and general-purpose registers, when available. Otherwise, the caller places them in the parameter area.

3. The caller places elements of all other types in general-purpose registers, when available. Otherwise, the caller places them in the parameter area.

■ Arguments that are part of a variable argument list are placed in general-purpose registers, when available. Otherwise, the caller places them in the parameter area.

> **Important:** When the return value of the called function would not be passed in registers, if it were passed as a parameter in a function call, the caller passes a pointer in GPR3 as an implicit first parameter of the called function. Therefore, the function's declared parameters start at GPR4. The pointer points to a section of memory large enough to hold the return value. See "Returning Results" (page 40) for more information.

> **Note:** Floating-point and vector elements passed by value are placed in floating-point registers and vector registers, respectively, which may differ from how they are passed in other binary interfaces.

Using ANSI C Prototypes

When the types of all the parameters of a subroutine are known at compile time, placing arguments into registers is straightforward.

For example, assume a routine calls the function `foo_ansi` declared like this:

```
int foo_ansi(int i, float f, long l, vector int v,
             double d, void* p, char c, short s);
```

The caller places the arguments to the function as shown in Table 5.

Table 5   Passing arguments to a function that declares all the types of its parameters

Argument	Type	Placed in	Reason
i	int	GPR3	Not a floating-point or vector element.
f	float	FPR1	First floating-point element, so it goes in the first floating-point register. GPR4 is skipped.
l	long	GPR5	Not a floating-point or vector element.
v	vector int	V2	First vector element, so it goes in the first vector register. No general-purpose register is skipped.
d	double	FPR2	Second floating-point element, so it goes in the next floating-point register available. GPR6 is skipped.
p	void*	GPR7	Not a floating-point or vector element.
c	char	GPR8	Not a floating-point or vector element.
s	short	GPR9	Not a floating-point or vector element.

Figure 3 illustrates the placement of arguments in registers and the parameter area.

Figure 3   Argument assignment when all parameter types are known
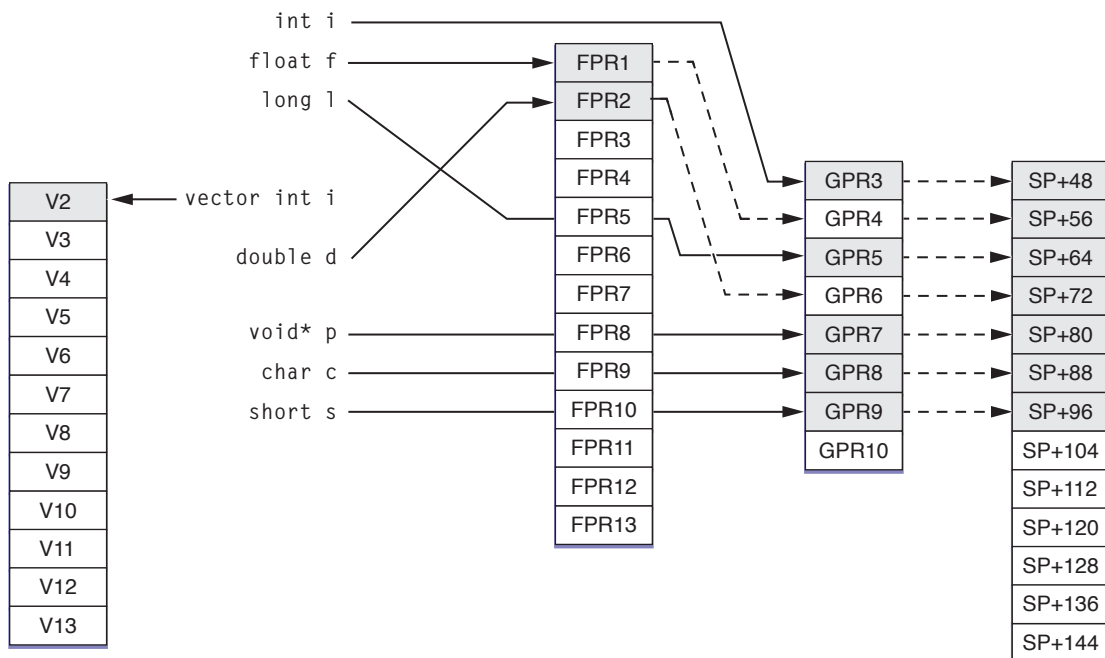
Using Structures

Assume the structure `data` and the function `bar` are declared like this:

```
struct data {
    float f;
    int i;
    double d;
    vector float v;
};
int bar(int a, struct data b, void* c);
```

Table 6 shows the register assignment when a routine calls `bar`.

Table 6   Passing arguments to a function with a `struct` parameter

Argument	Type	Placed in	Reason
a	int	GPR3	Not a floating-point or vector element.
b.f	float	FPR1	First floating-point element, so it goes in the first floating-point register. GPR4 is skipped. Because the b structure contains a vector, the entire struct needs 16-byte alignment in the parameter area.
b.i	int	GPR5 (low half)	Not a floating-point or vector element.
b.d	double	FPR2	Second floating-point element, so it goes in the next floating-point register available.

Argument	Type	Placed in	Reason
`b.v`	`vector float`	V2	First vector element, so it goes in the first vector register.
`c`	`void*`	GPR9	Not a floating-point or vector element.

Using Variable Argument Lists

Assume the structure `numbers` and the function `var` are declared like this:

```
struct numbers {
    float f;
    int i;
};
extern void var(int a, float b, vector float c, struct numbers n, ...);
```

Also assume a routine contains the following code:

```
int i1, i2;
float f1, f2;
vector float v1, v2;
struct numbers n1, n2;
...
var(i1, f1, v1, n1, i2, f2, v2, n2);
```

The caller assigns the arguments to `var` as shown in Table 7.

Table 7   Passing arguments to a function with a variable argument list

Argument	Type	Placed in	Reason
`i1`	`int`	GPR3	Not a floating-point or vector element.
`f1`	`float`	FPR1	First floating-point element, so it goes in the first floating-point register.
`v1`	`vector float`	V2	First vector element, so it goes in the first vector register.
`n1.f`	`float`	FPR2	Second floating-point element, so it goes in the next floating-point register available.
`n1.i`	`int`	GPR7 (low half)	Not a floating-point or vector element.
`i2`	`int` (unknown at compile time)	GPR8	A variable argument list element.
`f2`	`float` (unknown at compile time)	GPR9	A variable argument list element.
`v2`	`vector float` (unknown at compile time)	`SP+112` (16 bytes)	A variable argument list element and a vector. Must be 16-byte aligned; cannot use GPR10.

Argument	Type	Placed in	Reason
n2.f	float (unknown at compile time)	SP+128 (4 bytes)	A variable argument list element. No general-purpose registers available.
n2.i	int (unknown at compile time)	SP+132 (4 bytes)	A variable argument list element. No general-purpose registers available.

Using pre–ANSI C Prototypes

Assume the structure numbers and the function foo_pre_ansi are declared like this:

```
struct numbers {
    float f;
    int i;
};
void foo_pre_ansi();
```

Also assume a routine contains the following code:

```
...
int i;
float f;
vector float v;
struct numbers n;
...
foo_pre_ansi(i, f, v, n);
```

The caller assigns the arguments to foo_pre_ansi as shown in Table 8.

Table 8   Passing arguments to a function with a pre–ANSI C prototype

Argument	Type	Placed in	Reason
i	int (unknown at compile time)	GPR3	Not a floating-point or vector element.
f	float (unknown at compile time)	FPR1, GPR4	First floating-point element, so it goes in the first floating-point register and the next available general-purpose register.
v	vector float (unknown at compile time)	V2, GPR5–GPR6	First vector element, so it goes in the first vector register and next two general-purpose registers available.
n.f	float (unknown at compile time)	FPR2, GPR7 (high half)	Second floating-point element, so it goes in the next floating-point register available and the next general-purpose register available.
n.i	int (unknown at compile time)	GPR7 (low half)	Not a floating-point or vector element, so it goes in the next general-purpose register available.

Returning Results

A function result can be returned in registers or in memory, depending on the data type of the function's return value. When the return value of the called function would be passed in registers, if it were passed as a parameter in a function call, the called function places its return value in the same registers. Otherwise, the function places its result at the location pointed to by GPR3. See "Passing Arguments" (page 34) for more information.

Table 9 lists some examples of how return values can be passed to a calling routine.

Table 9   Examples of passing results to callers

Return type	Returned in
`int`	GPR3 (sign extended).
`unsigned short`	GPR3 (zero filled).
`long`	GPR3.
`long long`	GPR3.
`float`	FPR1.
`double`	FPR1.
`long double`	FPR1–FPR2.
`struct { float, float }`	FPR1, FPR2.
`struct { double, double }`	FPR1, FPR2.
`struct { long, long }`	GPR3, GPR4.
`struct { long[8] }`	GPR3, GPR4, ... GPR10.
`struct { long[10] }`	Memory location pointed to by GPR3, which is made up of 80 bytes of storage.
`vector float`	V2.
`complex float`	FPR1 (real number), FPR2 (imaginary number).
`complex double`	FPR1 (real number), FPR2 (imaginary number).
`complex long double`	FPR1–FPR2 (real number), FPR3–FPR4 (imaginary number).

Register Preservation

Table 10 lists the 64-bit PowerPC architecture registers used in this environment and their volatility in subroutine calls. Registers that must preserve their value after a function call are called **nonvolatile**.

Table 10   Processor registers in the 64-bit PowerPC architecture

Type	Name	Preserved	Notes
General-purpose register	GPR0	No	
	GPR1	Yes	Used as the stack pointer to store parameters and other temporary data items.
	GPR2	No	Available for general use.
	GPR3	No	The caller passes arguments to the called subroutine in GPR3 through GPR10. The caller may also pass the address to storage where the callee places its return value in this register.
	GPR4–GPR10	No	Used by callers to pass arguments to the called subroutine (see notes for GPR3).
	GPR11	Yes in nested functions. No in leaf functions.	In nested functions, the caller passes its stack frame to the nested function in this register. In leaf subroutines, the register is available. For details on nested functions, see the GCC documentation. This register is also used by lazy stubs in dynamic code generation to point to the lazy pointer.
	GPR12	No	Set to the address of the branch target before an indirect call for dynamic code generation. This register is not set for a subroutine that has been called directly, so subroutines that may be called directly should not depend on this register being set up correctly. See *Mach-O Programming Topics* for more information.
	GPR13	Yes	Reserved for thread-specific storage.
	GPR14–GPR31	Yes	
Floating-point register	FPR0	No	
	FPR1–FPR13	No	Used by callers to pass floating-point arguments to the called subroutine. Floating-point results are passed in FPR1.
	FPR14–FPR31	Yes	
Vector register	V0–V19	No	Callers use V2 through V13 to pass vector arguments to the called subroutine. Vector results are passed in V2.
	V20–V31	Yes	

Type	Name	Preserved	Notes
Special-purpose vector register	VRSAVE	Yes	32-bit special-purpose register. Each bit in this register indicates whether the corresponding vector register must be saved during a thread or process context switch.
Link register	LR	No	Stores the return address of the calling routine that called the current subroutine.
Count register	CTR	No	
Fixed-point exception register	XER	No	
Condition register fields	CR0, CR1	No	
	CR2–CR4	Yes	
	CR5–CR7	No	

IA-32 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 IA-32 environment most arguments are passed on the runtime stack; some vector arguments are 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 IA-32 architecture and whether their value is preserved after a subroutine call.

The function calling conventions used in the IA-32 environment are the same as those used in the System V IA-32 ABI, with the following exceptions:

- Different rules for returning structures

- The stack is 16-byte aligned at the point of function calls

- Large data types (larger than 4 bytes) are kept at their natural alignment

- Most floating-point operations are carried out using the SSE unit instead of the x87 FPU, except when operating on `long double` values. (The IA-32 environment defaults to 64-bit internal precision for the x87 FPU.)

The content of this article is largely based in *System V Application Binary Interface: Intel386 Architecture Processor Supplement*, available at http://www.caldera.com/developers/devspecs/abi386-4.pdf.

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

Data type	Size (in bytes)	Natural alignment (in bytes)
`unsigned short`	2	2
`signed short`	2	2
`unsigned int`	4	4
`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`	16	16
pointer	4	4

Table 2 shows the vector types available in this environment.

Table 2   Size and alignment of the vector types

Vector type	Element data type	Size (in bytes)	Alignment (in bytes)
`__m64`	`int`	8	8
`__m128i`	`int`	16	16
`__m128`	`float`	16	16
`__m128d`	`double`	16	16

These are some important details about this environment:

■   A byte is 8 bits long.

■   A null pointer has a value of 0.

■   This environment doesn't require 8-byte alignment for double-precision values.

■   This environment requires 16-byte alignment for 128-bit vector elements.

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

This section details 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.

Stack Structure

The IA-32 environment uses a stack that—at the point of function calls—is 16-byte aligned, grows downward, and contains local variables and a function's parameters. Each routine may add linkage information to its stack frame, but it's not required to do so. Figure 1 shows the stack before and during a subroutine call. (To help prevent the execution of malicious code on the stack, GCC protects the stack against execution.)

Figure 1 Stack layout
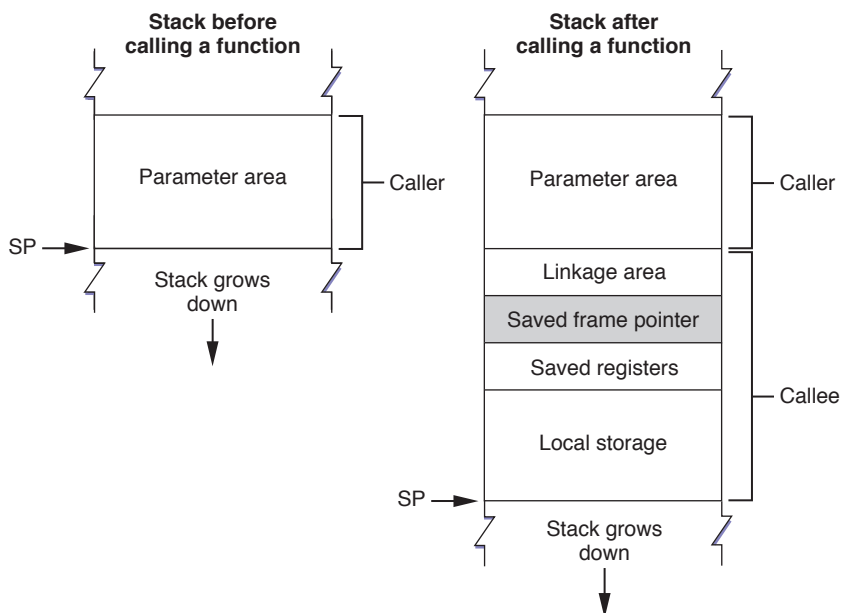
The **stack pointer** (SP) points to the bottom of the stack. Stack frames contain the following areas:

■ **The parameter area** stores the arguments the caller passes to the called subroutine. 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.

You can use the `gcc -fomit-frame-pointer` option to make the compiler not save, set up, and restore the frame pointer in function calls that don't need one, making the EBP register available for general use. However, doing so may impair debugging.

■ 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 50) 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 50) 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 stack frame pointer (EBP) onto the stack.

2. Sets the stack frame pointer to the value of the stack pointer (ESP).

3. Pushes the values of the registers that must be preserved (EDI, ESI, and EBX) onto the stack.

4. 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 (EDI, ESI, EBX, EBP) by popping the values saved on the stack by the prolog.

3. Returns.

> **Note:** Functions called during signal handling have no unusual restrictions on their use of registers. When a signal-handling function returns, the process resumes its original path with the registers restored to their original values.

Listing 1 shows the definition of the `simp` function.

Listing 1   Definition of the `simp` function

```
#include <stdio.h>
void simp(int i, short s, char c) {
```

```
    printf("Hi!\n");
}
```

Listing 2 shows a possible prolog for the `simp` function.

Listing 2   Example prolog

```
pushl   %ebp            ; save EBP
movl    %esp,%ebp       ; copy ESP to EBP
pushl   %ebx            ; save EBX
subl    $0x24,%esp      ; allocate space for local storage
```

Listing 3 shows a possible epilog for the `simp` function.

Listing 3   Example epilog

```
addl    $0x24,%esp      ; deallocate space for local storage
popl    %ebx            ; restore EBX
popl    %ebp            ; restore EBP
ret                     ; return
```

Passing Arguments

The compiler adheres to the following rules when passing arguments to subroutines:

1. The caller ensures that the stack is 16-byte aligned at the point of the function call.

2. The caller aligns nonvector arguments to 4-byte (32 bits) boundaries.

   The size of each argument is a multiple of 4 bytes, with tail padding when necessary. Therefore, 8-bit and 16-bit integral data types are promoted to 32-bit before they are pushed onto the stack.

3. The caller places arguments in the parameter area in reverse order, in 4-byte chunks. That is, the rightmost argument has the highest address.

4. The caller places all the fields of structures (or unions) with no vector elements in the parameter area. These structures are 4-byte aligned .

5. The caller places structures with vector elements on the stack, 16-byte aligned. Each vector within the structure is 16-byte aligned.

6. The caller places 64-bit vectors (`__m64`) on the parameter area, aligned to 8-byte boundaries.

7. The caller places vectors 128-bit vectors (`__m128`, `__m128d`, and `__m128i`) in registers XMM0 through XMM3). When the usable XMM registers are exhausted, the caller places 128-bit vectors in the parameter area. The caller aligns 128-bit vectors in the parameter area to 16-byte boundaries.

In general, the caller is responsible for removing all the parameters used in a function call after the called function returns. The only exception are parameters that are generated automatically by GCC. When a function returns a structure or union larger than 8 bytes, the caller passes a pointer to appropriate storage as the first argument to the function. GCC adds this parameter automatically in the code generated. See "Returning Results" (page 50) for more information. For example, the compiler would translate the code shown in Listing 4 to machine language as if it were written as shown in Listing 5.

Listing 4   Using a large structure—source code

```
typedef struct {
    float ary[8];
} big_struct;
big_struct callee(int a, float b) {
    big_struct callee_struct;
    ...
    return callee_struct;
}
caller() {
    big_struct caller_struct;
    caller_struct = callee(3, 42.0);
}
```

Listing 5   Using a large structure—compiler interpretation

```
typedef struct {
    float ary[8];
} big_struct;
void callee(big_struct *p, int a, float b)
{
    big_struct callee_struct;
    ...
    *p = callee_struct;
    return;
}
caller() {
    big_struct caller_struct;
    callee(&caller_struct, 3, 42.0);
}
```

Passing Arguments of the Fundamental Data Types

Assume the function `foo` is declared like this:

```
void foo(SInt32 i, float f, double d, SInt16 s, UInt8 c);
```

Figure 2 illustrates the placement of arguments in the parameter area at the point of the function call. Note the padding added to align the stack at 16 bytes.

Figure 2   Argument assignment with arguments of the fundamental data types
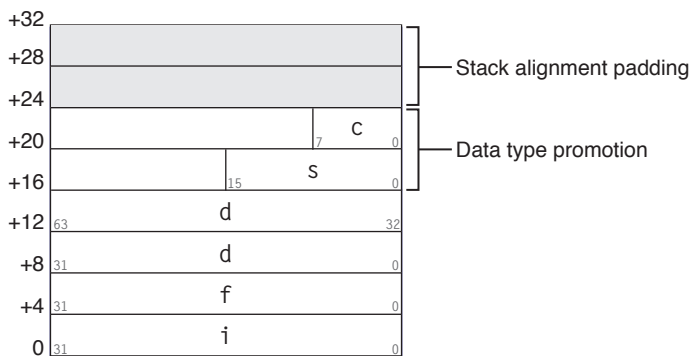
Passing Structures and Vectors

Assume the structure `data` and the function `bar` are declared like this:

```
struct data {
    float f;
    long long l;
    __m128 vf;
};
void bar(SInt32 i, UInt8 c, struct data b, __m128i vi, void* p);
```

Figure 3 illustrates the placement of arguments in the parameter area, the stack's 16-byte alignment padding at the point of the call, and the XMM registers.

Figure 3   Argument assignment with structure and vector arguments
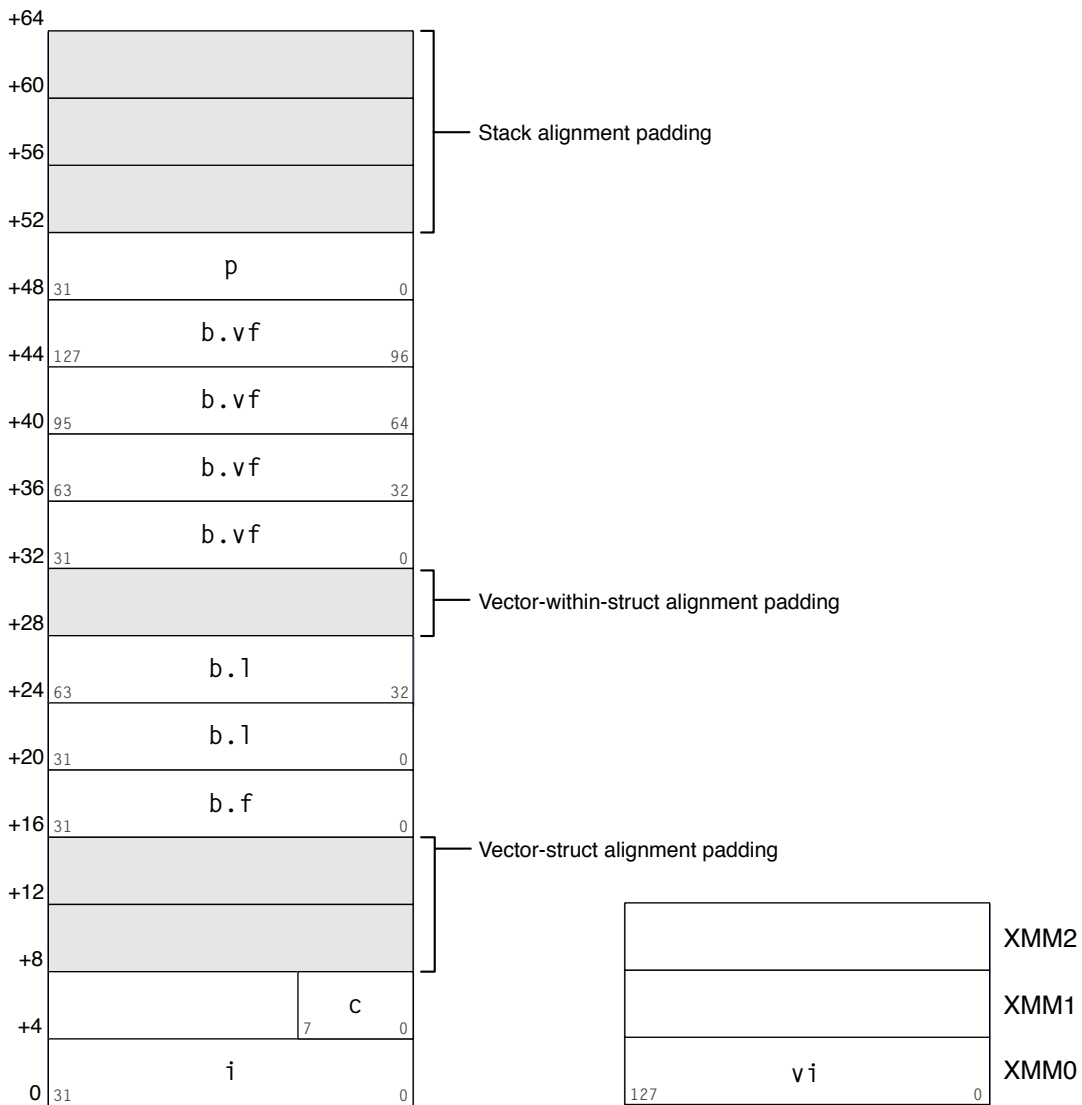
Returning Results

This is how functions pass results to their callers:

- **Scalar values.** Scalar values include structures that contain only one scalar value.

  - ❑ The called function places integral or pointer results in EAX.

  - ❑ The called function places floating-point results in ST0.

  - ❑ The caller removes the value from this register, even when it doesn't use the value.

- **Structures.** The called function returns structures according to their aligned size.

  - ❑ Structures 1 or 2 bytes in size are placed in EAX.

  - ❑ Structures 4 or 8 bytes in size are placed in: EAX and EDX.

  - ❑ Structures of other sizes are placed at the address supplied by the caller. For example, the C++ language occasionally forces the compiler to return a value in memory when it would normally be returned in registers. See "Passing Arguments" (page 47) for more information.

- **Vectors.**

  - ❑ The called function places vectors at the address supplied by the caller.

Register Preservation

Table 3 lists the IA-32 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 3   Processor registers in the IA-32 architecture

Type	Name	Preserved	Notes
General-purpose register	EAX	No	Used to return integral and pointer values. The caller may also place the address to storage where the callee places its return value in this register.
	EDX	No	Dividend register (divide operation). Available for general use for all other operations.
	ECX	No	Count register (shift and string operations). Available for general use for all other operations.
	EBX	Yes	Position-independent code base register. Available for general use in non–position-independent code.
	EBP	Yes	Stack frame pointer. Optionally holds the base address of the current stack frame. A routine's parameters reside in the previous frame as positive offsets of this register's value. Local variables reside at negative offsets.

Type	Name	Preserved	Notes
	ESI	Yes	Available for general use.
	EDI	Yes	Available for general use.
Stack-pointer register	ESP	Yes	Holds the address of the bottom of the stack.
Floating-point register	ST0	No	Used to return floating-point values. When the function doesn't return a floating-point value, this register must be empty. This register must also be empty on function entry.
	ST1–ST7	No	Available for general use. These registers must be empty on routine entry and exit.
64-bit register	MM0–MM7	No	Used to execute single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, 2-byte, and 4-byte integers.
128-bit register	XMM0–XMM7	No	Used to execute 32-bit and 64-bit floating-point arithmetic. Also used to execute single-instruction, multiple-data (SIMD) operations on 128-bit packed single-precision and double-precision scalar and floating-point values, and 128-bit packed byte, 2-byte, and 4-byte integers. XMM0–XMM3 are used to pass the first four vectors in a function call.
System-flags register	EFLAGS	No	Contains system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" direction (that is, 0) before entry to and upon exit from a routine. Other user flags have no specified role in the standard calling sequence and are not preserved.

x86-64 Function Calling Conventions

The Mac OS X x86-64 function calling conventions are the same as the function calling conventions described in *System V Application Binary Interface AMD64 Architecture Processor Supplement*, found at http://www.x86-64.org/documentation.html. See that document for details.

Other documents with information pertaining the Mac OS X x86-64 environment are:

■ *Mach-O Programming Topics*

■ *Mac OS X ABI Mach-O File Format Reference*

Document Revision History

This table describes the changes to *Mac OS X ABI Function Call Guide.*

Date	Notes
2009-02-04	Made content corrections.
2009-01-06	Made minor content changes.
	Corrected IA-32 function-result–return details.
2007-04-04	Added details about the Mac OS X x86-64 environment.
	Added cross-reference to System V x86-64 ABI document in "x86-64 Function Calling Conventions" (page 53).
2006-11-07	Clarified parameter-passing and floating-point operation details.
	Clarified how parameters are passed in the parameter area in the PPC32 environment in "Stack Structure" (page 13).
	Clarified how arrays and structures are placed in the parameter area in the PPC64 environment in "Stack Structure" (page 28).
	Indicated how floating point operations are performed in the IA-32 environment in "IA-32 Function Calling Conventions" (page 43).
	Clarified how structures are returned in the IA-32 environment in "Returning Results" (page 50).
2006-04-04	Corrected description of natural alignment in the PPC and PPC64 architectures, and clarified stack-alignment details in the IA32 architecture.
	Corrected the data types that yield better performance when using natural alignment in "32-bit PowerPC Function Calling Conventions" (page 9) and "64-bit PowerPC Function Calling Conventions" (page 25).
	Specified that function callers are responsible for aligning the stack at 16-byte boundaries at the point of function calls in "IA-32 Function Calling Conventions" (page 43).
2006-01-10	Specified when called functions remove parameters from the stack upon return in the Mac OS X IA-32 ABI.
	Updated "Passing Arguments" (page 47) and "Returning Results" (page 50) to describe how compiler-generated parameters are handled by called functions.

Date	Notes
2005-12-06	Changed the alignment values and red zone limits for 64-bit programs to their correct values.
2005-11-09	New document that describes the function-calling conventions used in the architectures supported by Mac OS X. Replaces information previously published in "PowerPC Runtime Architecture Guide."