Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, and so on. The system libraries described in Chapter 6 require this calling sequence.

NOTE

The standard calling sequence *requirements* apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nonetheless, it is recommended that all functions use the standard calling sequence when possible.

NOTE

C programs follow the conventions given here. For specific information on the implementation of C, see "Coding Examples" in this chapter.

Registers and the Stack Frame

The Intel386 architecture provides a number of registers. All the integer registers and all the floating-point registers are global to all procedures in a running program.

Brief register descriptions appear in Figure 3-14 more complete information appears later.

Figure 3-14: Processor Registers

Type	Name	Usage
General	`%eax`	Return value
	`%edx`	Dividend register (divide operations)
	`%ecx`	Count register (shift and string operations)
	`%ebx`	Local register variable
	`%ebp`	Stack frame pointer (optional)
	`%esi`	Local register variable
	`%edi`	Local register variable
	`%esp`	Stack pointer
Floating-point	`%st(0)`	floating-point stack top, return value
	`%st(1)`	floating-point next to stack top
	. . .	
	`%st(7)`	floating-point stack bottom

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Figure 3-15 shows the stack frame organization.

Figure 3-15: Standard Stack Frame

Position	Contents	Frame	
`4`*n*`+8(%ebp)`	argument word *n*	Previous	*High addresses*
	. . .		
`8(%ebp)`	argument word 0		
`4(%ebp)`	return address	Current	
`0(%ebp)`	previous `%ebp` (optional)		
`-4(%ebp)`	unspecified		
	. . .		
`0(%esp)`	variable size		*Low addresses*

Several key points about the stack frame deserve mention.

- The stack is word aligned. Although the architecture does not require any alignment of the stack, software convention and the operating system requires that the stack be aligned on a word boundary.

- Argument words are pushed onto the stack in reverse order (that is, the rightmost argument in C call syntax has the highest address), preserving the stack's word alignment. All incoming arguments appear on the stack, residing in the stack frame of the caller.
- An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the "unspecified" area of the standard stack frame.

All registers on the Intel386 are global and thus visible to both a calling and a called function. Registers `%ebp`, `%ebx`, `%edi`, `%esi`, and `%esp` "belong" to the calling function. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Some registers have assigned roles in the standard calling sequence:

`%esp` The *stack pointer* holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. At all times, the stack pointer should point to a word-aligned area.

`%ebp` The *frame pointer* optionally holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from `%ebp`, while local variables reside in the current frame, referenced as negative offsets from `%ebp`. A function must preserve this register's value for its caller. G

`%eax` *Integral and pointer return values* appear in `%eax`. A function that returns a `struct` or `union` value places the address of the result in `%eax`. Otherwise this is a scratch register.

`%ebx` As described below, this register serves as the *global offset table base register* for position-independent code. For absolute code, `%ebx` serves as a local register and has no specified role in the function calling sequence. In either case, a function must preserve the register value for the caller.

`%esi` and `%edi` These *local registers* have no specified role in the function calling sequence. A function must preserve their values for the caller.

`%ecx` and `%edx` *Scratch registers* have no specified role in the standard calling sequence. Functions do not have to preserve their values for the caller.

`%st(0)` *Floating-point return values* appear on the top of the floating-point register stack; there is no difference in the representation of single- or double-precision values in floating-point registers. If the function does not return a floating-point value, then this register must be empty. This register must be empty before entry to a function. G

`%st(1)` through `%st(7)` *Floating-point scratch registers* have no specified role in the standard calling sequence. These registers must be empty before entry and upon exit from a function.

EFLAGS The *flags register* contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" (that is, zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not preserved.

Floating-Point Control Word The Intel387 *control word* contains the floating-point flags, such as the rounding mode and exception masking. *

Signals can interrupt processes [see `signal`(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without the danger of signal handlers changing their values.

Functions Returning Scalars or No Value

A function that returns an integral or pointer value places its result in register `%eax`.

A floating-point return value appears on the top of the Intel387 register stack. The caller then must remove the value from the Intel387 stack, even if it doesn't use the value. Failure of either side to meet its obligations leads to undefined program behavior. The standard calling sequence does not include any method to detect such failures nor to detect return value type mismatches. Therefore the user must declare all functions properly. There is no difference in the representation of

single-, double- or extended-precision values in floating-point registers.

Functions that return no value (also called procedures or `void` functions) put no particular value in any register.

A `call` instruction pushes the address of the next instruction (the return address) onto the stack. The `ret` instruction pops the address off the stack and effectively continues execution at the next instruction after the `call` instruction. A function that returns a scalar or no value must preserve the caller's registers as described earlier. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (`%esp`) with the value it had before the `call` instruction was executed.

To illustrate, the following function prologue allocates 80 bytes of local stack space and saves the local registers `%ebx`, `%esi`, and `%edi`.

Figure 3-16: Function Prologue

```
prologue:
        pushl   %ebp            / save frame pointer
        movl    %esp, %ebp      / set new frame pointer
        subl    $80, %esp       / allocate stack space
        pushl   %edi            / save local register
        pushl   %esi            / save local register
        pushl   %ebx            / save local register
```

An epilogue for the example that restores the state for the caller. This example returns the value in `%edi` by moving it to `%eax`.

Figure 3-17: Function Epilogue

```
        movl    %edi, %eax      / set up return value
epilogue:
        popl    %ebx            / restore local register
        popl    %esi            / restore local register
        popl    %edi            / restore local register
        leave                   / restore frame pointer
        ret                     / pop return address
```

NOTE

Although some functions can be optimized to eliminate the save and restore of the frame pointer, the general case uses the standard prologue and epilogue.

Sections below describe where arguments appear on the stack. The examples are written as if the function prologue described above had been used.

Position-independent code uses the `%ebx` register to hold the address of the global offset table. If a function needs the global offset table's address, either directly or indirectly, it is responsible for computing the value. See "Coding Examples" later in this chapter and "Dynamic Linking" in Chapter 5 for more information.

Functions Returning Structures or Unions

If a function returns a structure or union, then the caller provides space for the return value and places its address on the stack as argument word zero. In effect, this address becomes a "hidden" first argument. Having the caller supply the return object's space allows re-entrancy.

NOTE

Structures and unions in this context have fixed sizes. The ABI does not specify how to handle variable sized objects.

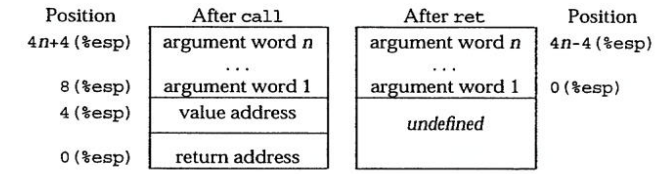
A function that returns a structure or union also sets %eax to the value of the original address of the caller's area before it returns. Thus when the caller receives control again, the address of the returned object resides in register %eax and can be used to access the object. Both the calling and the called functions must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame;
- The called function must use the address from the frame and copy the return value to the object so supplied;
- The called function must remove this address from the stack before returning.

Failure of either side to meet its obligations leads to undefined program behavior. The standard function calling sequence does not include any method to detect such failures nor to detect structure and union type mismatches. Therefore the user must declare all functions properly.

Figure 3-18 illustrates the stack contents when the function receives control (after the call instruction) and when the calling function again receives control (after the ret instruction).

Figure 3-18: Stack Contents for Functions Returning `struct/union`



To illustrate, the following function prologue allocates 80 bytes of local stack space and saves the local registers %ebx, %esi, and %edi. Additionally, it removes the "hidden" argument from the stack and saves it in the highest word of the local stack frame.

Figure 3-19: Function Prologue (Returning `struct/union`**)**

```
prologue:
        popl    %eax            / pop return address
        xchgl   %eax, 0(%esp)   / swap return address
                                  / and return value address
        pushl   %ebp            / save frame pointer
        movl    %esp, %ebp      / set new frame pointer
        subl    $80, %esp       / allocate local space
        pushl   %edi            / save local register
        pushl   %esi            / save local register
        pushl   %ebx            / save local register
        movl    %eax, -4(%ebp)  / save return value address
```

An epilogue for the example that restores the state for the caller.

Figure 3-20: Function Epilogue

```
epilogue:
        movl    -4(%ebp), %eax      / set up return value
        popl    %ebx                / restore local register
        popl    %esi                / restore local register
        popl    %edi                / restore local register
        leave                       / restore frame pointer
        ret                         / pop return address
```

NOTE

Although some functions can be optimized to eliminate the save and restore of the frame pointer, the general case uses the standard prologue and epilogue.

Sections below describe where arguments appear on the stack. The examples are written as if the function prologue described above had been used.

Position-independent code uses the `%ebx` register to hold the address of the global offset table. If a function needs the global offset table's address, either directly or indirectly, it is responsible for computing the value. See "Coding Examples" later in this chapter and "Dynamic Linking" in Chapter 5 for more information.

Integral and Pointer Arguments

As mentioned, a function receives all its arguments through the stack; the last argument is pushed first. In the standard calling sequence, the first argument is at offset `8(%ebp)`, the second argument is at offset `12(%ebp)`, and so on. Functions pass all integer-valued arguments as words, expanding or padding signed or unsigned bytes and halfwords as needed.

Figure 3-21: Integral and Pointer Arguments

Call	Argument	Stack address
`g(1, 2, 3, (void *)0);`	`1`	`8(%ebp)`
	`2`	`12(%ebp)`
	`3`	`16(%ebp)`
	`(void *)0`	`20(%ebp)`

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one word, double-precision use two, and extended-precision use three. See "Coding Examples" for information about floating-point arguments and variable argument lists. The example below uses only double-precision arguments. Single- and extended-precision arguments behave as specified above.

Figure 3-22: Floating-Point Arguments

Call	Argument	Stack address
`h(1.414, 1, 2.998e10);`	word 0, `1.414`	`8(%ebp)`
	word 1, `1.414`	`12(%ebp)`
	`1`	`16(%ebp)`
	word 0, `2.998e10`	`20(%ebp)`
	word 1, `2.998e10`	`24(%ebp)`

NOTE

The Intel386 architecture does not require doubleword alignment for double-precision values. Nevertheless, for data structure compatibility with other Intel architectures, compilers may provide a method to align double-precision values on doubleword boundaries.

CAUTION

A compiler that provides the doubleword alignment mentioned above would have to maintain doubleword alignment for the stack. Moreover, the arguments in the preceding example would appear in different positions. Programs built with the doubleword alignment facility would not conform to the Intel386 ABI, and their function calling sequence would not be compatible with conforming Intel386 programs.

Structure and Union Arguments

As described in the data representation section, structures and unions can have byte, halfword, or word alignment, depending on the constituents. An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument. To ensure that data in the stack is properly aligned, the stack pointer should always point to a word boundary. Structure and union arguments are pushed onto the stack in the same manner as integral arguments, described above. This provides call-by-value semantics, letting the called function modify its arguments without affecting the calling function's object.

Figure 3-23: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	`1`	`8(%ebp)`
	word 0, `s`	`12(%ebp)`
	word 1, `s`	`16(%ebp)`
	...	...

DRAFT COPY
March 19, 1997
File: abi_386/chap3 (Delta 44.6)
386:adm.book:sum

Page: 45