Chapter 17

High-Level Language Interface

Objectives

- To review motivation for writing mixed-mode programs
- To discuss the principles of mixed-mode programming
- To describe how assembly language procedures are called from C
- To illustrate how C functions are called from assembly language procedures
- To explain how inline assembly language code is written

Thus far, we have written standalone assembly language programs. This chapter considers mixed-mode programming, which refers to writing parts of a program in different programming languages. We use C and Pentium assembly languages to illustrate how such mixed-mode programs are written. The motivation for mixed-mode programming is discussed in Section 17.1. Section 17.2 gives an overview of mixed-mode programming, which can be done either by inline assembly code or by separate assembly modules. The inline assembly method is discussed in Section 17.5. Other sections focus on the separate assembly module method.

Section 17.3 describes the mechanics involved in calling assembly language procedures from a C program. This section presents details about parameter passing, returning values of C functions, and so on. Section 17.4 shows how a C function can be called from an assembly language procedure. The last section summarizes the chapter.

17.1 Why Program in Mixed Mode?

In this chapter we focus on mixed-mode programming that involves C and assembly languages. Thus, we write part of the program in C and the other part in the Pentium assembly language. We use the `gcc` compiler and NASM assembler to explain the principles involved in mixed-mode programming. This discussion can be easily extended to a different set of languages and compilers/assemblers.

In Chapter 1 we discussed several reasons why one would want to program in the assembly language. Although it is possible to write a program entirely in the assembly language, there are several disadvantages in doing so. These include

- Low productivity
- High maintenance cost
- Lack of portability

Low productivity is due to the fact that assembly language is a low-level language. As a result, a single high-level language instruction may require several assembly language instructions. It has been observed that programmers tend to produce the same number of lines of debugged and tested source code per unit time irrespective of the level of the language used. As the assembly language requires more lines of source code, programmer productivity tends to be low.

Programs written in the assembly language are difficult to maintain. This is a direct consequence of it's being a low-level language. In addition, assembly language programs are not portable. On the other hand, the assembly language provides low-level access to system hardware. In addition, the assembly language may help us reduce the execution time.

As a result of these pros and cons, some programs are written in mixed mode using both high-level and low-level languages. System software often requires mixed-mode programming. In such programs, it is possible for a high-level procedure to call a low-level procedure, and vice versa. The remainder of the chapter discusses how mixed-mode programming is done in C and assembly languages. Our goal is to illustrate only the principles involved. Once these principles are understood, the discussion can be generalized to any type of mixed-mode programming.

17.2 Overview

There are two ways of writing mixed-mode C and assembly programs: inline assembly code or separate assembly modules. In the inline assembly method, the C program module contains assembly language instructions. Most C compilers including `gcc` allow embedding assembly language instructions within a C program by prefixing them with **asm** to let the compiler know that it is an assembly language instruction. This method is useful if you have only a small amount of assembly code to embed. Otherwise, separate assembly modules are preferred. We discuss the inline assembly method in Section 17.5.

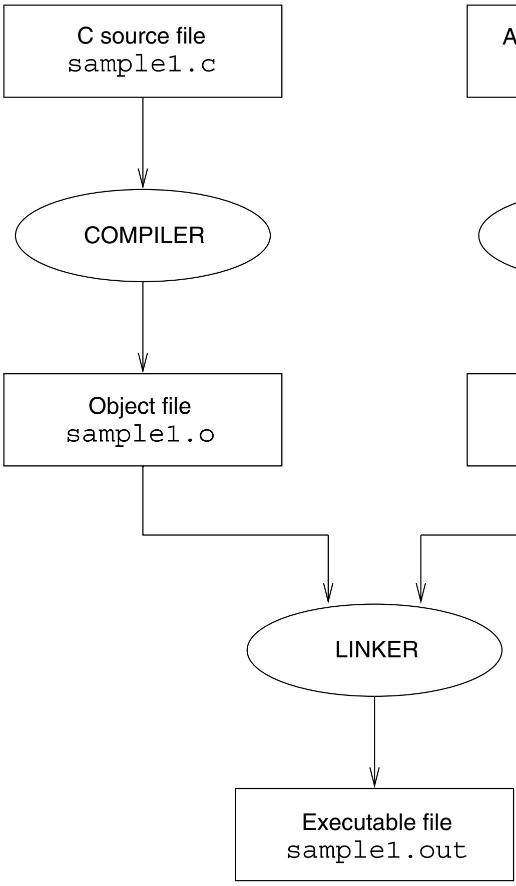

Figure 17.1 Steps involved in compiling mixed-mode programs.

When separate modules are used for C and assembly languages, each module can be translated into the corresponding object file. To do this translation, we use a C compiler for the C modules and an assembler for the assembly modules, as shown in Figure 17.1. Then the linker can be used to produce the executable file from these object files.

Suppose our mixed-mode program consists of two modules:

- One C module, file `sample1.c`, and
- One assembly module, file `sample2.asm`.

The process involved in producing the executable file is shown in Figure 17.1. We can invoke the NASM assembler as

```
nasm -f elf sample2.asm
```

This creates the `sample2.o` object file. We can compile and link the files with the following command:

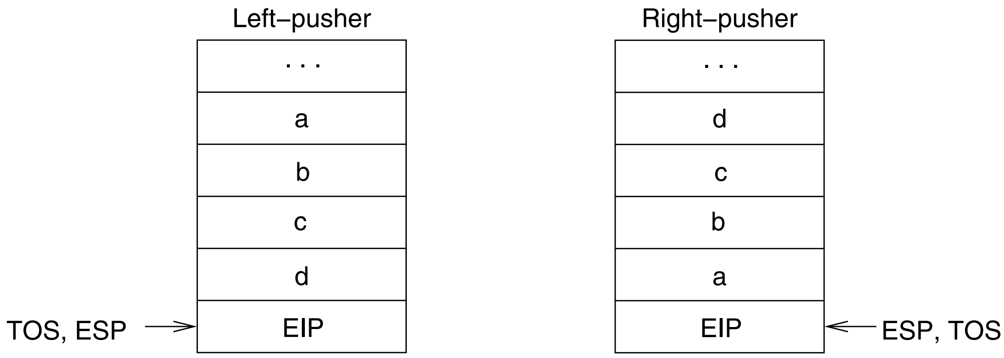

Figure 17.2 Two ways of pushing arguments onto the stack.

```
gcc -o sample1.out sample1.c sample2.o
```

This command instructs the compiler to first compile `sample1.c` to `sample1.o`. The linker is automatically invoked to link `sample1.o` and `sample2.o` to produce the executable file `sample1.out`.

17.3 Calling Assembly Procedures from C

Let us now discuss how we can call an assembly language procedure from a C program. The first thing we have to know is what communication medium is used between the C and assembly language procedures, as the two procedures may exchange parameters and results. You are right if you guessed it to be the stack.

Given that the stack is used for communication purposes, we still need to know a few more details as to how the C function places the parameters on the stack, and where it expects the assembly language procedure to return the result. In addition, we should also know which registers we can use freely without worrying about preserving their values. Next we discuss these issues in detail.

Parameter Passing

There are two ways in which arguments (i.e., parameter values) are pushed onto the stack: from left to right or from right to left. Most high-level languages push the arguments from left to right. These are called *left-pusher* languages. C, on the other hand, pushes arguments from right to left. Thus, C is a *right-pusher* language. The stack state after executing

```
sum(a,b,c,d)
```

is shown in Figure 17.2. From now on, we consider only right-pushing of arguments, as we focus on the C language.

To see how `gcc` pushes arguments onto the stack, take a look at the following C program (this is a partial listing of Example 17.1):

```
int main(void)
{
    int     x=25, y=70;
    int     value;
    extern  int test(int, int, int);

    value = test (x, y, 5);
    . . .
}
```

The assembly language translation of the procedure call (use `-S` option to generate the assembly source code) is shown below:[1]

```
    push    5
    push    70
    push    25
    call    test
    add     ESP,12
    mov     [EBP-12],EAX
```

This program is compiled with `-O2` optimization. This optimization is the reason for pushing constants 70 and 25 instead of variables `x` and `y`. If you don't use this optimization, `gcc` produces the following code:

```
    push    5
    push    [EBP-8]
    push    [EBP-4]
    call    test
    add     ESP,12
    mov     [EBP-12],EAX
```

It is obvious from this code fragment that the compiler assigns space for variables `x`, `y`, and `value` on the stack at EBP−4, EBP−8, and EBP−12, respectively. When the `test` function is called, the arguments are pushed from right to left, starting with the constant 5. Also notice that the stack is cleared of the arguments by the C program after the call by the following statement:

```
add   ESP,12
```

So, when we write our assembly procedures, we should not bother clearing the arguments from the stack as we did in our programs in the previous chapters. This convention is used because C allows a variable number of arguments to be passed in a function call (see our discussion in Section 5.8 on page 146).

[1]Note that `gcc` uses AT&T syntax for the assembly language—not the Intel syntax we have been using in this book. To avoid any confusion, the contents are reported in our syntax. The AT&T syntax is introduced in Section 17.5.

Returning Values

We can see from the assembly language code given in the last subsection that the EAX register is used to return the function value. In fact, the EAX is used to return 8-, 16-, and 32-bit values. To return a 64-bit value, use the EDX:EAX pair with the EDX holding the upper 32 bits.

We have not discussed how floating-point values are returned. For example, if a C function returns a `double` value, how do we return this value? We discuss this issue in Chapter 18.

Preserving Registers

In general, the called assembly language procedure can use the registers as needed, except that the following registers should be preserved:

```
EBP, EBX, ESI, EDI
```

The other registers, if needed, must be preserved by the calling function.

Globals and Externals

Mixed-mode programming involves at least two program modules: a C module and an assembly module. Thus, we have to declare those functions and procedures that are not defined in the same module as external. Similarly, those procedures that are accessed by another module should be declared as global, as discussed in Chapter 5. Before proceeding further, you may want to review the material on multimodule programs presented in Chapter 5 (see Section 5.10 on page 156). Here we mention only those details that are specific to the mixed-mode programming involving C and assembly language.

In most C compilers, external labels should start with an underscore character (_). The C and C++ compilers automatically append the required underscore character to all external functions and variables. A consequence of this characteristic is that when we write an assembly procedure that is called from a C program, we have to make sure that we prefix an underscore character to its name. However, `gcc` does not follow this convention by default. Thus, we don't have to worry about the underscore.

17.3.1 Illustrative Examples

We now look at three examples to illustrate the interface between C and assembly language programs. We start with a simple example, whose C part has been dissected before.

Example 17.1 *Our first mixed-mode example.*

This example passes three parameters to the assembly language function `test1`. The C code is shown in Program 17.1 and the assembly code in Program 17.2. The function `test1` is declared as external in the C program (line 12) and global in the assembly program (line 8). Since C clears the arguments from the stack, the assembly procedure uses a simple `ret` to transfer control back to the C program. Other than these differences, the assembly procedure is similar to several others we have written before.

Program 17.1 An example illustrating assembly calls from C: C code (in file `hll_ex1c.c`)

```
/*****************************************************************
 * A simple program to illustrate how mixed-mode programs are
 * written in C and assembly languages. The main C program calls
 * the assembly language procedure test1.
 *****************************************************************/
#include        <stdio.h>

int main(void)
{
    int    x = 25, y = 70;
    int    value;
    extern int test1 (int, int, int);

    value = test1(x, y, 5);
    printf("Result = %d\n", value);

    return 0;
}
```

Program 17.2 An example illustrating assembly calls from C: assembly language code (in file `hll_test.asm`)

```
;-----------------------------------------------------------
; This procedure receives three integers via the stack.
; It adds the first two arguments and subtracts the third one.
; It is called from the C program.
;-----------------------------------------------------------
segment .text

global  test1

test1:
      enter   0,0
      mov     EAX,[EBP+8]        ;  get argument1 (x)
      add     EAX,[EBP+12]       ;  add argument 2 (y)
      sub     EAX,[EBP+16]       ;  subtract argument3 (5)
      leave
      ret
```

Example 17.2 *An example to show parameter passing by call-by-value as well as call-by-reference.*

This example shows how pointer parameters are handled. The C main function requests three integers and passes them to the assembly procedure. The C program is given in Program 17.3. The assembly procedure `min_max`, shown in Program 17.4, receives the three integer values and two pointers to variables `minimum` and `maximum`. It finds the minimum and maximum of the three integers and returns them to the main C function via these two pointers. The minimum value is kept in EAX and the maximum in EDX. The code given on lines 27 to 30 in Program 17.4 stores the return values by using the EBX register in the indirect addressing mode.

Program 17.3 An example with the C program passing pointers to the assembly program: C code (in file `hll_minmaxc.c`)

```
/*****************************************************************
 * An example to illustrate call-by-value and                    *
 * call-by-reference parameter passing between C and             *
 * assembly language modules. The min_max function is            *
 * written in assembly language (in the file hll_minmaxa.asm).   *
 *****************************************************************/
#include <stdio.h>
int main(void)
{
      int     value1, value2, value3;
      int     minimum, maximum;
      extern  void min_max (int, int, int, int*, int*);

      printf("Enter number 1 = ");
      scanf("%d", &value1);
      printf("Enter number 2 = ");
      scanf("%d", &value2);
      printf("Enter number 3 = ");
      scanf("%d", &value3);

      min_max(value1, value2, value3, &minimum, &maximum);
      printf("Minimum = %d, Maximum = %d\n", minimum, maximum);
      return 0;
}
```

Program 17.4 An example with the C program passing pointers to the assembly program: assembly language code (in file `hll_minmax_a.asm`)

```
;-------------------------------------------------------------
; Assembly program for the min_max function - called from the
; C program in the file hll_minmaxc.c. This function finds
; the minimum and maximum of the three integers it receives.
;-------------------------------------------------------------
global  min_max

min_max:
      enter   0,0
      ; EAX keeps minimum number and EDX maximum
      mov     EAX,[EBP+8]      ; get value 1
      mov     EDX,[EBP+12]     ; get value 2
      cmp     EAX,EDX          ; value 1 < value 2?
      jl      skip1            ; if so, do nothing
      xchg    EAX,EDX          ; else, exchange
skip1:
      mov     ECX,[EBP+16]     ; get value 3
      cmp     ECX,EAX          ; value 3 < min in EAX?
      jl      new_min
      cmp     ECX,EDX          ; value 3 < max in EDX?
      jl      store_result
      mov     EDX,ECX
      jmp     store_result
new_min:
      mov     EAX,ECX
store_result:
      mov     EBX,[EBP+20]     ; EBX = &minimum
      mov     [EBX],EAX
      mov     EBX,[EBP+24]     ; EBX = &maximum
      mov     [EBX],EDX
      leave
      ret
```

Example 17.3 *Array sum example.*

This example illustrates how arrays, declared in C, are accessed by assembly language procedures. The array `value` is declared in the C program, as shown in Program 17.5 (line 12). The assembly language procedure computes the sum as shown in Program 17.6. As in the other programs in this chapter, the C program clears the parameters off the stack. We will redo this example using inline assembly in Section 17.5.

Program 17.5 An array sum example: C code (in file `hll_arraysumc.c`)

```c
/**********************************************************
 * This program reads 10 integers into an array and calls an
 * assembly language program to compute the array sum.
 * The assembly program is in the file "hll_arraysuma.asm".
 **********************************************************/
#include        <stdio.h>

#define  SIZE  10

int main(void)
{
    int    value[SIZE], sum, i;
    extern int array_sum(int*, int);

    printf("Input %d array values:\n", SIZE);
    for (i = 0; i < SIZE; i++)
        scanf("%d",&value[i]);

    sum = array_sum(value,SIZE);
    printf("Array sum = %d\n", sum);

    return 0;
}
```

Program 17.6 An array sum example: assembly language code (in file `hll_arraysuma.asm`)

```asm
;------------------------------------------------------------
; This procedure receives an array pointer and its size via
; the stack. It computes the array sum and returns it.
;------------------------------------------------------------
segment .text

global  array_sum

array_sum:
      enter   0,0
      mov     EDX,[EBP+8]     ; copy array pointer to EDX
      mov     ECX,[EBP+12]    ; copy array size to ECX
      sub     EBX,EBX         ; array index = 0
      sub     EAX,EAX         ; sum = 0 (EAX keeps the sum)
```

```
add_loop:
        add     EAX,[EDX+EBX*4]
        inc     EBX             ; increment array index
        cmp     EBX,ECX
        jl      add_loop
        leave
        ret
```

17.4 Calling C Functions from Assembly

So far, we have considered how a C function can call an assembler procedure. Sometimes it is desirable to call a C function from an assembler procedure. This scenario often arises when we want to avoid writing assembly language code for a complex task. Instead, a C function could be written for those tasks. This section illustrates how we can access C functions from assembly procedures. Essentially, the mechanism is the same: we use the stack as the communication medium, as shown in the next example.

Example 17.4 *An example to illustrate a C function call from an assembly procedure.*

In previous chapters, we used simple I/O routines to facilitate input and output in our assembly language programs. If we want to use the C functions like `printf()` and `scanf()`, we have to pass the arguments as required by the function. In this example, we show how we can use these two C functions to facilitate input and output of integers. This discussion can be generalized to other types of data.

Here we compute the sum of an array passed onto the assembly language procedure `array_sum`. This example is similar to Example 17.3, except that the C program does not read the array values; instead, the assembly program does this by calling the `printf()` and `scanf()` functions as shown in Program 17.8. In this program, the prompt message is declared as a string on line 9 (including the newline). The assembly language version implements the equivalent of the following `printf` statement we used in Program 17.5:

```
printf("Input %d array values:\n", SIZE);
```

Before calling the `printf` function on line 21, we push the array size (which is in ECX) and the string onto the stack. The stack is cleared on line 22.

The array values are read using the read loop on lines 26 to 36. It uses the `scanf` function, the equivalent of the following statement:

```
scanf("%d",&value[i]);
```

The required arguments (array and format string pointers) are pushed onto the stack on lines 28 and 29 before calling the `scanf` function on line 30. The array sum is computed using the add loop on lines 41 to 45 as in Program 17.6.

Program 17.7 An example to illustrate C calls from assembly programs: C code (in file `hll_arraysum2c.c`)

```
/**********************************************************
 * This program calls an assembly program to read the array
 * input and compute its sum. This program prints the sum.
 * The assembly program is in the file "hll_arraysum2a.asm".
 **********************************************************/
#include        <stdio.h>

#define  SIZE  10

int main(void)
{
    int    value[SIZE];
    extern int array_sum(int*, int);

    printf("sum = %d\n",array_sum(value,SIZE));

    return 0;
}
```

Program 17.8 An example to illustrate C calls from assembly programs: assembly language code (in file `hll_arraysum2a.asm`)

```
;------------------------------------------------------------
; This procedure receives an array pointer and its size
; via the stack. It first reads the array input from the
; user and then computes the array sum.
; The sum is returned to the C program.
;------------------------------------------------------------
segment .data
scan_format    db   "%d",0
printf_format  db   "Input %d array values:",10,13,0

segment .text

global  array_sum
extern  printf,scanf

array_sum:
      enter   0,0
```

```
        mov     ECX,[EBP+12]    ; copy array size to ECX
        push    ECX             ; push array size
        push    dword printf_format
        call    printf
        add     ESP,8           ; clear the stack

        mov     EDX,[EBP+8]     ; copy array pointer to EDX
        mov     ECX,[EBP+12]    ; copy array size to ECX
read_loop:
        push    ECX             ; save loop count
        push    EDX             ; push array pointer
        push    dword scan_format
        call    scanf
        add     ESP,4           ; clear stack of one argument
        pop     EDX             ; restore array pointer in EDX
        pop     ECX             ; restore loop count
        add     EDX,4           ; update array pointer
        dec     ECX
        jnz     read_loop

        mov     EDX,[EBP+8]     ; copy array pointer to EDX
        mov     ECX,[EBP+12]    ; copy array size to ECX
        sub     EAX,EAX         ; EAX = 0 (EAX keeps the sum)
add_loop:
        add     EAX,[EDX]
        add     EDX,4           ; update array pointer
        dec     ECX
        jnz     add_loop
        leave
        ret
```

17.5 Inline Assembly

In this section we look at writing inline assembly code. In this method, we embed assembly language statements within the C code. We identify assembly language statements by using the `asm` construct. (You can use `__asm__` if `asm` causes a conflict, e.g., for ANSI C compatibility.)

We now have a serious problem: the syntax that the `gcc` compiler uses for assembly language statements is different from the syntax we have been using so far. We have been using the Intel syntax (NASM, TASM, and MASM use this syntax). The `gcc` compiler uses the AT&T syntax, which is used by GNU assemblers. It is different in several aspects from the Intel syntax. But don't worry! We give an executive summary of the differences so that you can understand the syntactical differences without spending too much time.