CHAPTER 10

Mixing C and Assembly

Occasionally it is required to write a program in both C and assembly language. There are several possible reasons.

- First, an experienced programmer might want to optimize a performance-critical function manually in assembly, instead of relying on compilers. Many profiling tools can identify the most time-consuming functions and compilers often have limited intelligence in optimizing these functions. A handcrafted assembly code can out-perform high-level languages, such as C.
- Second, writing a program in assembly allows a programmer to use processor-specific instructions. For example, a test-and-set atomic assembly instruction can implement locks and semaphores. Another example is that most C compilers do not use some operations available on Cortex-M processors, such as `ROR` (rotate right) and `RRX` (rotate right extended).
- Third, assembly programs can directly access hardware, which is especially useful for device drivers and processor booting code.

> *"The good thing about standards is that there are so many to choose from."*
>
> **Andrew Tanenbaum, famous computer scientisit**

The embedded application binary interface (EABI) briefly introduced in Chapter 8.2 defines low-level standards of interfacing program modules that are compiled separately, no matter whether these modules are written in C or assembly. The EABI specifies (1) standards for data types, data alignments, and executable file formats, and (2) conventions for function calls, parameter passing, registers usage, and stack frame. If a program is written in C, compilers ensure that these standards are followed strictly. However, if a program is developed in assembly, it is

the programmer's responsibility to adhere to these standards. The standard allows programmers to mix C and assembly in the application.

10.1 Data Types and Access

While the size of a basic data type in the C language depends on the compilers and platforms, the following table lists the typical size of commonly used data types in the C language.

Data Type	Size (bits)	Alignment	Data Range
bool	8	byte	0 or 1. Bits 1 – 7 are ignored
char	8	byte	-128 to 127 (signed), or 0 to 255 (unsigned)
int	32	word	-2,147,483,648 to 2,147,483,647 (signed), or 0 to 4,294,967,296 (unsigned)
short int	16	halfword	-32,768 to 32,767 (signed), or 0 to 65,536 (unsigned)
long int	32	word	same as int
long long	64	word	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed), or 0 to 18,446,744,073,709,551,616 (unsigned)
float	32	word	+/- (1.4023×10^{-45} to $3.4028 \times 10^{+38}$), always signed
double	64	word	+/- (4.9406×10^{-324} to 1.7977×10^{308}), always signed
long double	96	word	very large range
pointer	32	word	0 to 4,294,967,296

Table 10-1. Data size and alignment of basic data types in C

10.1.1 Signed or Unsigned Integers

When programming in an assembly language, it is the programmer's responsibility to interpret whether a data item is signed or unsigned. For example, when loading an 8-bit data into a 32-bit register, the program should use `LDRSB` (load register with signed byte) to access a signed character and `LDRB` (load register with byte) to retrieve an unsigned character.

- `LDRSB` loads a byte from the memory into a register and performs sign extension. The sign extension duplicates the sign bit of the 8-bit data to all bits at the most significant side of a register to preserve the positive or negative sign.
- `LDRB` loads a byte from the memory into a register and simply pads the left of the register with zeros.

For example, when a program loads an 8-bit binary data `0x88` (+136 for unsigned or -120 for signed) from the memory into a 32-bit register, should the register be `0xFFFFFF88` or

0x00000088? It depends on the programmer's intention. If these 8 bits represent a signed number, LDRSB should be used to preserve the sign. If they represent an unsigned number, LDRB should be used. Similarly, LDRSH (load register with signed halfword) and LDRH (load register with halfword) bring a 16-bit signed and unsigned number into a register, respectively. Table 10-2 summarizes these load instructions.

Variable	Instruction	Description	Sign Extension
unsigned char	LDRB	Load register with byte	No
unsigned short	LDRH	Load register with halfword	No
unsigned/signed int	LDR	Load register with word	No
char	LDRSB	Load register with signed byte	Yes
short	LDRSH	Load register with signed halfword	Yes

Table 10-2. ARM assembly instructions for accessing various basic integer data types

Correspondingly, STRB (store register byte) and STRH (store register halfword) store either a signed number or an unsigned number into the memory. Loading or storing a 32-bit integer does not need to take care of the sign because each register has the same amount of bits as the integer.

A 64-bit integer takes two registers, and it can be loaded by using two separate LDR instructions or a single LDRD (load registers with double words).

C Program	Assembly Program

signed long long x = -1;	LDR r3, =x ; load memory address of x
	LDRD r0, r1, [r3] ; r0 lower word, r1 higher word
	
	x DCW 0xFFFFFFFF, 0xFFFFFFFF ; allocate 8 bytes

Example 10-1. Loading a 64-bit integer from the memory

10.1.2 Data Alignment

Most computer systems have some alignment requirement on the starting memory address of a variable. The memory address of a C variable often has to be aligned, as listed in Table 10-1. The smallest unit exchanged between the processor and the memory is a byte (8 bits), and thus the memory address is always in terms of bytes.

A variable is n-byte aligned in memory if its starting memory address is some multiple of n. Typically, *n* is a power of 2, such as 2 (halfword aligned), 4 (word aligned), and 8 (double word aligned). Suppose a 32-bit variable is word aligned. If the address of the next available byte in memory is 0x8001, the variable is then stored in a continuous span of 4 bytes from 0x8004 to 0x8007. The compiler or the program inserts three meaningless

bytes at memory addresses `0x8001`, `0x8002`, and `0x8003`. These three bytes are called *padding bytes*.

Enforcing data alignment is to improve the memory performance. A memory system consists of multiple storage units, and the processor typically distributes data among these units in a round-robin fashion. Because the number of pins available on a processor is limited, these memory units have to share some pins in the memory address bus. To allow these memory units to transfer data concurrently, the target data stored in all memory units needs to share a portion of their memory addresses. The data alignment ensures that all data of a variable stored in different memory units meet this requirement. When the processor reads a properly aligned variable, only one access is required to transfer the data out of these memory units. Otherwise, two separate memory accesses might be necessary, slowing down the processor performance.

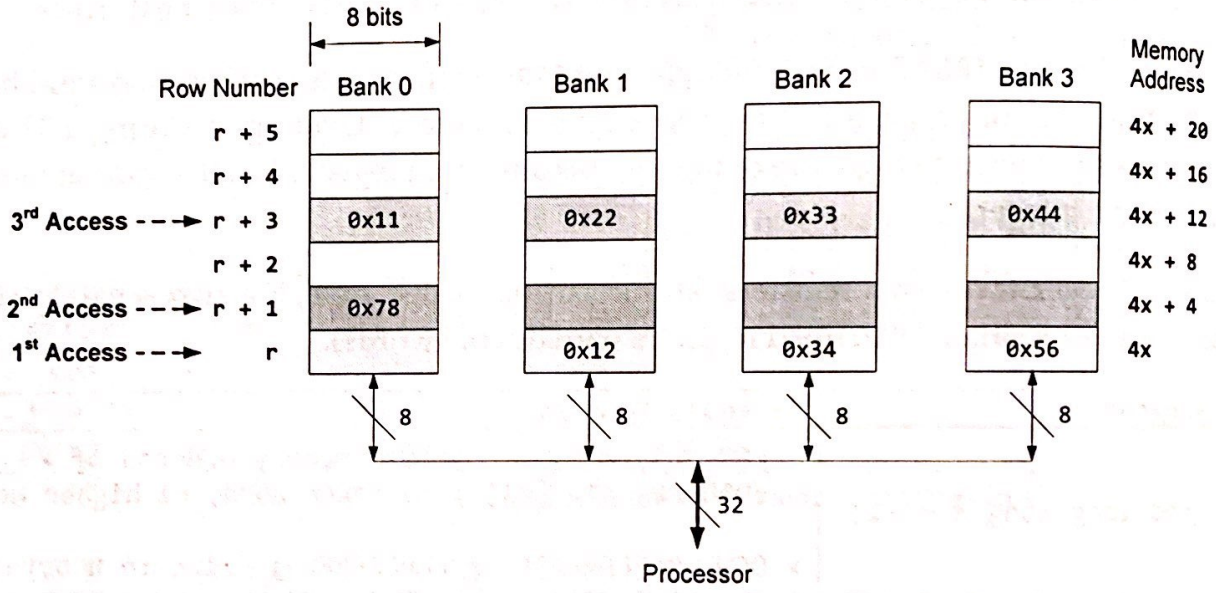

Figure 10-1. Loading unaligned data `0x78563412` takes two accesses even if the processor allows unaligned memory accesses. Loading aligned data `0x44332211` takes only one access.

As shown in Figure 10-1, the data memory is organized into four banks, and these banks can feed the 32-bit data bus. Four bytes in the same row of all banks can be loaded into the processor concurrently. In this example, data `0x78563412` is not aligned with word boundaries, and the processor takes two memory accesses to load the data `0x78563412` to a register. However, it takes only one memory access to load data `0x44332211`.

As introduced in Chapter 3.6, the "`ALIGN`" directive gives data alignment requirements to compilers. The syntax is "`ALIGN boundary, offset`". The boundary is any power of 2. The default boundary is 4, making the next variable align to a word boundary. The offset specifies how many bytes the next variable should start from the word boundary. The default offset is `0`.

```
  AREA myData, DATA, ALIGN=2
  ; word aligned

a DCB  0x11

  ALIGN 4
  ; word aligned
b DCD  0x12345678

  ALIGN 4,3
  ; word aligned with an offset of 3
c DCB  0x22

  ALIGN 4,2
  ; word aligned with an offset of 2
d DCW  0xAABB
```

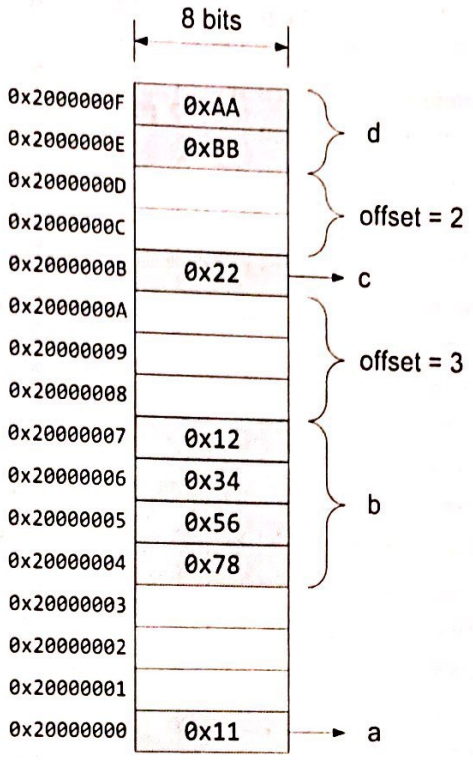

Figure 10-2. Memory layout

10.1.3 Data Structure Padding

A data structure defined in C language aggregates multiple basic variables into a single complex entity. By default, compilers ensure that all variables in a structure are aligned to their required memory boundaries. In a structure array, compilers also ensure that all variables in this array meet their alignment requirements. Therefore, compilers may place padding bytes between structure variables.

C language also supports *packed structures* in which variables are not aligned. Therefore, compilers do not add any padding bytes into a data structure. Packed structures are often used in communication protocols (such as USB) to save transmission time.

Figure 10-3 and Figure 10-4 compare the memory layout of an unpacked structure defined in Example 10-2.

Unpacked Structure	Packed Structure

`struct Position {`
`  char x;`
`  char y;`
`  char x;`
`  int time;`
`  short scale;`
`} array[2];`
	`__packed struct Position {`
	`  char x;`
	`  char y;`
	`  char x;`
	`  int time;`
	`  short scale;`
	`} array[2];`

Example 10-2. Comparison of unpacked and packed structure in C

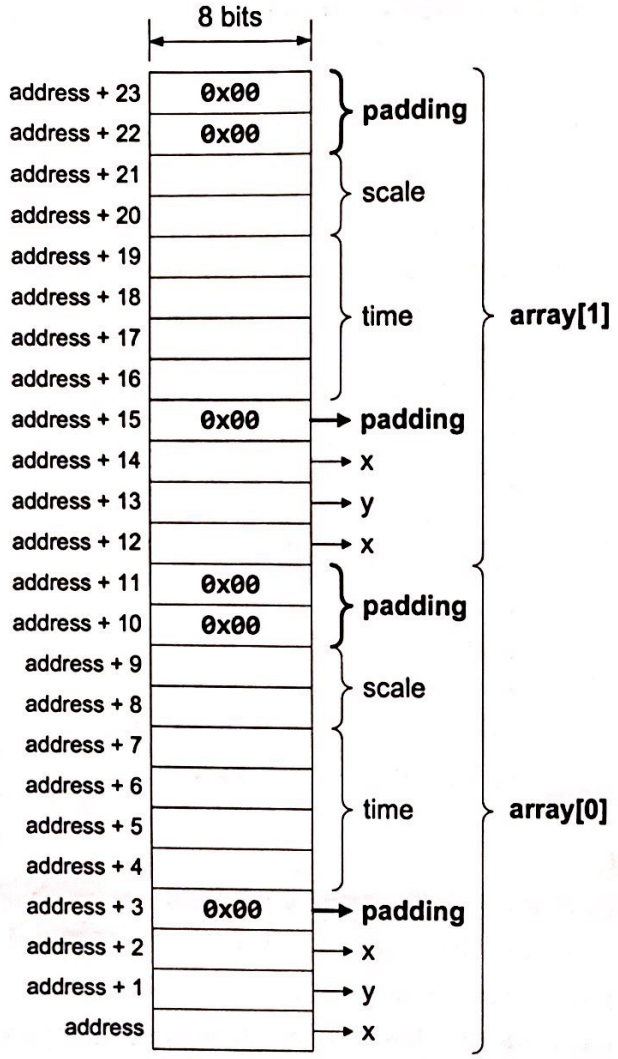

Figure 10-3. In an unpacked structure, variables are aligned. Specifically the integer variable and the structure are aligned in words.

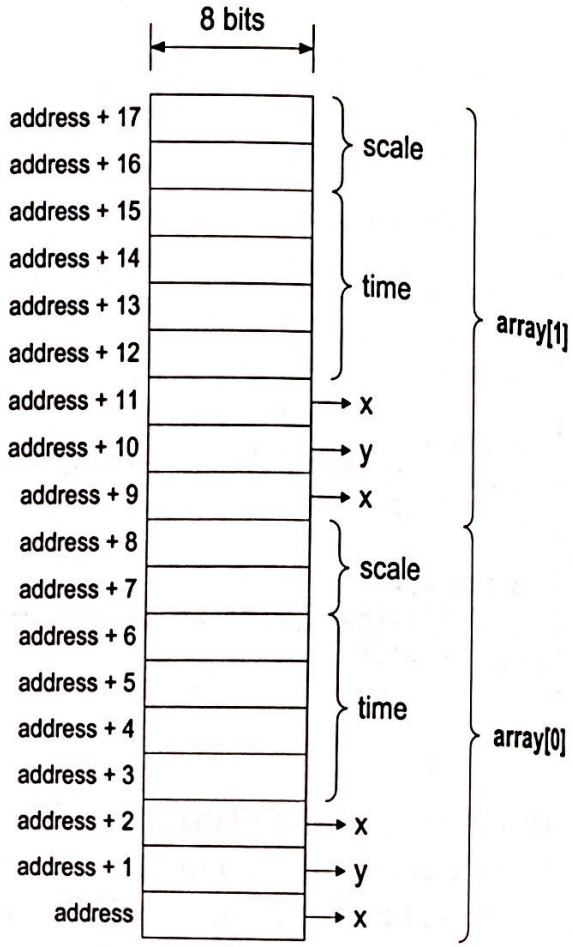

Figure 10-4. In a packed structure, variables are not aligned. Cortex-M processors support unaligned access in LDR/STR, LDRT/STRT, LDRH/STRH, and LDRHT/STRHT.

In Figure 10-3, the compiler inserts three padding bytes into the structure position.

- The first padding byte is inserted after variable *x* to make the next integer variable *time* aligned to some word boundary.
- In a structure array, the compiler also ensures that all variables in this array meet their alignment requirements. Therefore, two additional bytes are added at the end of the data structure, making the size of the *Position* structure a multiple of four. This padding also makes the variables in this array, particularly the time variable, align properly.

In fact, two structure definitions give below are equivalent. Compilers allocate proper padding bytes in an unpacked structure.

```
struct Position {
   char x;
   char y;
   char x;
   int time;
   short scale;
} array[2];
```

Equivalent ⟺

```
struct Position {
   char x;
   char y;
   char x;
   char padding_1;
   int time;
   short scale;
   char padding_2[2];
} array[2];
```

Example 10-3. Compiler inserts three padding bytes to an unpacked structure

In Figure 10-4, the position structure uses the type modifier "**__packed**" to the compiler to produce an unaligned memory layout. Specifically, the integer variable *time* is not aligned to a word boundary, and the short variable *scale* is not aligned to a halfword boundary. Therefore, there is no padding between structure members or at the end of the structure. The **__unpack** modifier is often used to map a structure to a special data area in memory, such as a USB communication package received in a memory buffer.

Programs often do not use packed structures. No ARM processors released before ARM-V6 support unaligned memory accesses. The instruction "`LDR r1, [r0]`" would generate an alignment exception if the memory address stored in `r0` were not a multiple of four. The Cortex-M processors do support unaligned memory accesses. However, unaligned accesses are still slower than aligned memory accesses, and thus it is recommended to avoid using unaligned accesses.

Packed structures and unpacked structures are not compatible with each other. We cannot assign or cast one to the other. The only way to assign a packed structure to an unpacked structure is to copy all structure members one by one.

Suppose we want to set *`array[0].time`* to 1234. Example 10-4 compares the assembly codes that update the *time* variable of the unpacked and packed structure respectively.

Unpacked Structure	Packed Structure

`LDR r0, =array ; load base address`
`LDR r1, [r0, #4] ; array[0].time`
`LDR r2, #1234`
`STR r2, [r0, #4] ; array[0].time`
	`LDR r0, =array ; load base address`
	`LDR r1, [r0, #3] ; array[0].time`
	`LDR r2, #1234`
	`STR r2, [r0, #3] ; array[0].time`

Example 10-4. Accessing members of an unpacked and packed structure in assembly

In the unpacked structure, the access to *`array[0].time`* is aligned. However, in the packed structure, the access is misaligned. Even though the misaligned accesses "`LDR r1, [r0, #3]`" and "`STR r2, [r0, #3]`" are supported in Cortex-M, their access speed is slower than the aligned accesses "`LDR r1, [r0, #4]`" and "`STR r2, [r0, #4]`".

10.2 Special Variables

This section discusses two special types of variables in C: static variables and volatile variables.

10.2.1 Static Variables

> *A* `static` *variable is initialized only once. Its lifetime is across its entire program runtime.*

Different from local variables, a ***static*** C variable has a lifetime over the entire program runtime. A static variable declared within a C function is initialized only once at the compiling time no matter how many times this function is called. This static variable is visible only inside this function.

A static variable can be either global or local. A static local variable can only be visitable or available within the scope of the function in which this variable is declared. A static global variable can only be accessed within the source file in which it was declared, and other source files cannot access it.

> ***Always avoid global variables.***

A static variable is preferred to a global variable in C because a local or global static variable has a narrower access range. A program should avoid global variables whenever possible. A global variable is accessible to all source codes in any file of a software system. The major problem of using global variables is that they create hidden coupling between different software modules that is difficult to identify and understand, thus increasing the risk of software bugs. Because of the implicit interference created by global variables, a bug in one software module might cause the failure of another seemly-unrelated module, making the debug process difficult.

One effective way to avoid global variables is to use static variables. As shown in Example 10-5, the counter is declared as a global static variable, instead of a global variable. Therefore, the counter can only be accessed within in that source file. Codes in other source files cannot access the counter variable.

If a global variable is only accessed by one subroutine, the program may declare this variable as a local static variable within that subroutine. The access scope of a local static variable is the subroutine that declares it. When that subroutine is called successively, the value of the local static variable is retained.

```
static int counter = 0;

void increase(void){
  counter++;
}

void decrease(void){
  counter--;
}
```

Example 10-5. Example of using a global static variable

Example 10-6 and Example 10-7 compare how a local static variable and a local regular variable are accessed in assembly. All static variables are allocated in the data memory. However, a local variable is often stored in a register or the heap region of the data memory. A static variable is always loaded from the memory first, and then is stored back to the memory before exiting the subroutine. Therefore, if the subroutine is called again, the static variable keeps its previous value, instead of its initial value.

Example of a local non-static variable

C Program

```
int foo();

int main(void) {
   int y;
   y = foo();  // y = 6
   y = foo();  // y = 6
   y = foo();  // y = 6
   while(1);
}

int foo() {
   int x = 5;  // x is a local variable
   x = x + 1;
   return(x)
}
```

Assembly Program

```
        AREA static_demo, CODE
        EXPORT __main
        ALIGN
        ENTRY

__main  PROC
        BL  foo      ; r0 = 6
        BL  foo      ; r0 = 6
        BL  foo      ; r0 = 6
stop    B   stop
        ENDP

foo     PROC
        MOV r0, #5
        ADD r0, r0, #1
        BX  lr
        ENDP
        END
```

Example 10-6. If x is not declared as static, *foo*() always returns the same value.

In the program given in Example 10-6, variable *x* is not declared as static. Thus, *foo* returns the same value each time it is called. From the assembly implementation, we can notice that the local variable x is always reinitialized when *foo* is called. Plus, in this example, x is stored in a register and its value is lost (not saved in the data memory) after *foo* exits.

Example of a local static variable

C Program

```
int foo();

int main(void) {
  int y;
  y = foo();  // y = 6
  y = foo();  // y = 7
  y = foo();  // y = 8
  while(1);
}

int foo() {

  // local static variable
  // x is initialized only once
  static int x = 5;


  x = x + 1;
  return(x)
}
```

Assembly Program

```
        AREA myData, DATA
        ALIGN
        // Reserve space for x
x       DCD   5

        AREA static_demo, CODE
        EXPORT __main
        ALIGN
        ENTRY

__main  PROC
        BL   foo    ; r0 = 6
        BL   foo    ; r0 = 7
        BL   foo    ; r0 = 8
stop    B    stop
        ENDP


foo     PROC
        ; load address of x
        LDR  r1, =x
        ; load value of x
        LDR  r0, [r1]
        ADD  r0, r0, #1
        ; save value of x
        STR  r0, [r1]
        BX   lr
        ENDP
        END
```

Example 10-7. When x is declared as static, the *foo*() function returns different values.

In the program given in Example 10-7, variable *x* is declared as static locally within the *foo* function.

- The local static variable *x* is only initialized once. The initialization is carried out at compile time instead of at runtime. As you see from the assembly code, the variable is defined in the data region with an initial value of 5. No matter how many times *foo* runs, variable *x* is never re-initialized.
- When *foo* needs to increase the value of the local static variable *x*, the value of variable x is read from the memory at the beginning of *foo* and is saved into the memory before *foo* exits. Therefore, *foo* returns a different result each time it runs. On the contrary, *foo* in Example 10-6, in which *x* is not static, always returns the same value.

Example 10-8 gives another example of using static variables. The program uses the static variable *sum* to check whether an integer number is a palindrome number. A palindrome number remains the same if all digits are reversed.

C Program

```
int isPal(int);


int main(){
    int n;
    n = isPal(12321);
    while(1);
}


// Check palindrome number
int isPal(int n){
  static int sum = 0;
  int r;
  if(n!=0) {

    r = n % 10;




    sum = sum*10 + r;


    isPal (n/10);
  }



  if (sum == n)
    return 1;
  else
    return 0;
}
```

Assembly Program

```
        AREA myData, DATA
        ALIGN
sum     DCD 0

        AREA palindrome, CODE
        EXPORT __main
        ALIGN
        ENTRY
__main  PROC
        LDR   r0, =12321
        BL    isPal
stop    B     stop
        ENDP

; Recursively check palindrome
isPal   PROC
        PUSH  {r4, lr}
        MOV   r4, r0
        CBZ   r4, done        ; if n is 0, done
        MOV   r2, #10
        SDIV  r1, r4, r2      ; r1 = n/10
        MLS   r3, r1, r2, r4 ; r3 = n - r1 * 10;
        LDR   r1, =sum
        LDR   r1, [r1]          ; r1 = sum
        ADD   r1, r1,r1,LSL #2 ; r1 = 5*sum
        ADD   r1, r3,r1,LSL #1 ; sum = sum*10+r;
        LDR   r2, =sum
        STR   r1, [r2]          ; save sum
        MOV   r2, #10
        SDIV  r0, r4, r2        ; r0 = n/10
        BL    isPal             ; recursive call
done    LDR   r1, =sum
        LDR   r1, [r1]

        CMP   r1, r4
        BNE   no
yes     MOV   r0, #1            ; if palindrome
        B     exit
no      MOV   r0, #0            ; if not palindrome
exit    POP   {r4, pc}
        ENDP
        END
```

Example 10-8. Example of using a local static variable *sum* in a function

10.2.2 Volatile Variables

When the compiler optimizes a C program, a hard-to-find hidden error is that the program mistakenly reuses the value of a variable stored in a register, instead of reloading it from memory each time. In order to avoid such compilation error, the program should declare the variable as volatile, such as:

```
volatile int variable;
```

The keyword volatile forces the compiler to generate an executable, which always loads the variable value from the memory whenever this variable is read, and always stores the variable in the memory whenever it is written.

A variable declared as **volatile** *indicates that this variable may be changed by an input, not by the program. Therefore, the program should always re-read its value from the memory if the variable is used.*

Example 10-10 gives a simple example to illustrate the necessity of declaring a variable counter, shared by two concurrently running tasks (*main* function and *SysTick_Handler*), as volatile. In this example, the main program uses the SysTick to implement a time delay. It sets up the SysTick timer and then waits until the SysTick interrupt service routine reduces the counter to 0. The SysTick decrements the counter by one when a system timer interrupt occurs. Chapter 11.4 gives the implementation of *SysTick_Init*().

Main Program (*main.c*)

```
// volatile unsigned int counter;
unsigned int counter;
extern void task();
extern void SysTick_Init();

int main(void) {
   counter = 10;
   SysTick_Init();
   while(counter != 0);  // Delay
   // Continue the task
   while(1);
}
```

Interrupt Service Routine (*isr.s*)

```
        AREA ISR, CODE, READONLY
        IMPORT counter
        ENTRY

SysTick_Handler PROC
        EXPORT SysTick_Handler
        LDR r1, =counter
        LDR r0, [r1]     ; load counter
        SUB r0, r0, #1   ; counter--
        STR r0, [r1]     ; save counter
        BX  LR           ; exit
        ENDP
        END
```

Example 10-9. A C variable is not declared as *volatile* while it should be.

Compilers often attempt to optimize the program, but sometimes can cause troubles. The compiler observes that, after the counter is initialized to 10, the value of the counter variable is not modified directly by *main*() or indirectly by any subroutine called from *main*(). Loading data from memory is much slower than retrieving data from registers.

Therefore, the compiler may choose to reuse the value of the counter stored in a register, instead of fetching the counter value again from the memory, when the counter is accessed in the while loop. Example 10-10 compares the assembly program generated by the compiler when the counter variable is declared as volatile or non-volatile.

- If the counter is not declared as volatile, the while loop is a dead loop. *SysTick_Handler* periodically decrements the counter and stores its value in the memory. However, the main program repeatedly checks register `r0`, without reloading the latest value of the counter from the memory.
- If the counter is declared as volatile, the dead loop problem is avoided.

If counter is not declared as volatile

```
__main PROC
       LDR r1, =counter
       MOV r0, #10
       STR r0, [r1]
       BL  SysTick_Init

wait   CMP r0, #0 ; r0 does not hold
                  ; latest counter value
       BNE wait   ; Thus, a dead loop
stop   B   stop
       ENDP
```

If counter is declared as volatile

```
__main PROC
       LDR r1, =counter
       MOV r0, #10
       STR r0, [r1]
       BL  SysTick_Init

wait   LDR r1, =counter
       LDR r0, [r1]
       CMP r0, #0
       BNE wait
stop   B   stop
       ENDP
```

Example 10-10. Comparison of assembly instructions generated by the compiler when the counter variable is declared as volatile and non-volatile.

A C program should declare any variable that represents the data of a memory-mapped I/O register as `volatile`. Memory-mapped I/O has been widely used to access peripheral devices. Data and control registers of external devices are mapped to specific memory addresses, and a program can use memory pointers to access these hardware registers, such as the following C statement.

```
unsigned int *p = (unsigned int *) 0x60002400;
```

To prevent the compiler from optimizing out these memory pointers incorrectly, these pointers must be declared as volatile. The following example uses a memory pointer to access a 32-bit hardware register mapped to the memory address `0x60002400`.

```
volatile unsigned int *p = (unsigned int *) 0x60002400;
```

In sum, a variable should be declared as volatile to prevent the compiler from optimizing it away when (1) this variable is updated by external memory-mapped hardware, or (2) this variable is global and is changed by interrupt handlers or by multiple threads.

10.3 Inline Assembly

A block of assembly code, called inline assembly, can be directly embedded in a C program. It is convenient for programmers because it does not require different assemble and link processes. Another advantage of inline assembly is that it can flexibly access C variables without export and import operations, which would be needed if the assembly code were written as an assembly subroutine.

10.3.1 Assembly Functions in a C Program

When a C program declares a function with "__asm", the assembly code in this function has to preserve the environment.

When a block of assembly code is embedded within a function by using "__asm", the assembly code does not need to preserve the environment.

A C program can have inline assembly by using the "__asm" keyword. It has two different uses. The first is to specify a function that is implemented in assembly completely. The second is to specify multiple lines of assembly code within a C function.

When a function is declared with "__asm", the assembly implementation must preserve the runtime environment via the stack and recover the environment before exiting from the subroutine. The assembly code can directly access the registers and must follow the procedure call protocol. Example 10-11 and Example 10-12 use "__asm" to implement a C function in assembly.

```
__asm int sum4(int a, int b, int c, int d){
      ; arguments stored in r0, r1, r2, r3
      PUSH {r4, lr}       ; preserve environment in stack
      MOV r4, r0          ; r0 = 1st argument
      ADD r4, r4, r1      ; r1 = 2nd argument
      ADD r4, r4, r2      ; r2 = 3rd argument
      ADD r0, r4, r3      ; r3 = 4th argument, r0 = return
      POP {r4, pc}        ; recover environment from stack
}

int main(void){
      int s = sum4(1, 2, 3, 4);
      while(1);
}
```

Example 10-11. Using inline assembly to implement a subroutine that adds four integers.

```
char a[25] = "Hello!";
char b[25];

__asm void strcpy(char *src, char *dst){
loop   LDRB r2, [r0], #1   ; 1st argument, r0 = src, post-index
       STRB r2, [r1], #1   ; 2nd argument, r1 = dst, post-index
       CMP  r2, #0
       BNE  loop
       BX   lr
}
int main(void){
      strcpy(a, b);
      while(1);
}
```

Example 10-12. Using inline assembly to copy a string.

When a function is declared with "__asm", the compiler only creates the interface of this function and does not provide any actual implementation. Therefore, Example 10-11 uses PUSH and POP to preserve and recover the running environment of the caller.

10.3.2 Inline Assembly Instructions in a C Program

When "__asm" is used to declare a block of assembly instructions in a C function, the assembly code cannot access registers and does not need to preserve the runtime environment in the stack. The compiler automatically generates necessary code to preserve the environment.

Additionally, the assembly code treats each C variable as a register. These C variables are called *virtual registers* and they can be accessed in assembly instructions. Compilers replace virtual registers with real registers.

Also, the comments of the assembly code have to be in C style. Example 10-13 has a block of assembly instructions in a C function.

```
int sum4(int a, int b, int c, int d){
     int t;
     __asm {
         ADD t, a, b;  // t, a, and b are virtual registers
         ADD t, c;     // Cannot directly access r0 - r15
         ADD t, d;     // Have to use comment style of C
     }
     return t;
}
int main(void){
     int s = sum4(1, 2, 3, 4);
     while(1);
}
```

Example 10-13. Using "__asm" to declare a block of assembly instructions in a C function.

10.4 Calling Assembly Subroutines from a C Program

A large software project often has its program code saved in multiple small source files, instead of a single monolithic file. This technique not only improves the software modularity and maintainability, but also reduces the compilation time. These files can be compiled separately so that unmodified files do not need to be recompiled.

This section shows how a C program calls assembly subroutines that are in separate source files.

- The assembly code must use the directive "EXPORT" or "GLOBAL" to make all variables or subroutines that are accessed in the C program as global. These directives make subroutine names visible outside this source code module. Consequently, the compiler can locate them when linking the object files generated from their source codes.
- The C program must declare these functions by using the keyword "extern".

10.4.1 Example of Calling an Assembly Subroutine

In the following example, the C program calls the assembly subroutine *strlen*, which calculates the length of a string. The C program and the assembly program are in two separate source files: *main.c* and *strlen.s*.

C Program (*main.c*)

```
char str[25] = "Hello!";

extern int strlen(char* s);

int main(void){
    int i;
    i = strlen(str);
    while(1);
}
```

Assembly Program (*strlen.s*)

```
        AREA stringLength, CODE
        EXPORT strlen       ; make strlen visible
        ALIGN
strlen  PROC
        PUSH {r4, lr}       ; preserve r4 and lr
        MOV  r4, #0         ; initialize length
loop    LDRB r1, [r0, r4]   ; r0 = string address
        CBZ  r1, exit       ; branch if zero
        ADD  r4, r4, #1     ; length++
        B    loop           ; do it again
exit    MOV  r0, r4         ; place result in r0
        POP  {r4, pc}       ; exit
        ENDP
```

Example 10-14. A C program calls an assembly routine stored in a different file.

- The assembly subroutine follows the procedure call protocol defined in ARM embedded application binary interface (EABI) and assumes argument *str* is passed in register `r0`. Furthermore, the caller expects that the assembly subroutine returns a 32-bit result in register `r0` and a 64-bit result in registers `r1:r0`.

- The C program declares the assembly function to be called by using the keyword "extern" to inform the compiler that the implementation of this function is in another file.
- The assembly subroutine uses "EXPORT strlen" to make the symbol strlen visible to the linker. Note all symbols are case-sensitive.

10.4.2 Example of Accessing C Variables in Assembly

An assembly program can access global variables defined in a C program or a separate assembly source file. When an assembly program accesses a global variable defined elsewhere, it needs to import that variable name by using the directive "IMPORT". An imported variable name, or called a symbol, is resolved at link time. In the following example, the C program declares the global variable counter. The assembly code uses "IMPORT counter" to access this global variable.

C Program (*main.c*)

```
int counter;

extern int getValue();
extern void setValue(int c);

void increment();

int main(void) {
   int c = 0;
   setValue(1);
   increment();
   c = getValue();
   while(1);
}

void increment(){
   counter += 1;
}
```

Assembly Program (*count.s*)

```
            AREA count, CODE
            IMPORT counter
            ALIGN

setValue    PROC
            EXPORT setValue
            LDR r1, =counter
            STR r0, [r1]
            BX  lr
            ENDP

getValue    PROC
            EXPORT getValue
            LDR r1, =counter
            LDR r0, [r1]
            BX  lr
            ENDP

increment   PROC
            EXPORT increment [WEAK]
            LDR r1, =counter
            LDR r0, [r1]
            ADD r0, r0, #1
            STR r0, [r1]
            BX  lr
            ENDP
            END
```

Example 10-15. Example of accessing a C variable in assembly routines

The assembly program exports the *increment* symbol with weak specified. By default, all exporting statements are strong. At the linking stage, a strongly exported symbol replaces

a weakly exported symbol of the same name. The linker reports a fatal error if there more than one strong instance of the same symbol name. Because the symbol *increment* is exported weakly and the one in the C program is exported strongly, the *increment* function defined in C overrides the one defined in the assembly. Consequently, when the *increment* function is called, the *counter* variable is incremented by two, instead of one.

10.5 Calling C Functions from Assembly Programs

An assembly program can call functions implemented in C. The assembly program needs to follow the procedure call protocol defined in ARM embedded application binary interface (EABI).

- Specifically, the assembly program needs to place the input arguments of a C function in registers `r0-r1` before it calls the function.
- The assembly program also expects that the result is returned in register `r0` if the C function returns a value less than 32 bits.
- If the result has more than 32 bits, registers `r0 – r4` can be used to hold the result.

10.5.1 Example of Calling a C Function

In the following example, the assembly program calls the *strlen* function implemented in C. The C function returns the length of the string in register `r0` to the assembly program. In the assembly code, the C function names have to be imported to avoid linking errors.

Assembly Program (*main.s*)

```
        AREA my_strlen, CODE
        EXPORT __main
        IMPORT strlen
        ALIGN
        ENTRY

__main  PROC
        LDR   r0, =str
        BL    strlen
stop    B     stop
        ENDP

        AREA myData, DATA
        ALIGN
Str     DCB   "12345678",0
        END
```

C Program (*strlen.c*)

```
int strlen(char *s){
   int i = 0;

   while( s[i] != '\0' )
      i++;

   return i;
}
```

Example 10-16. Example of an assembly program that calls a C subroutine

If "WEAK" is specified in the import directive, the linker does not produce any error if the symbol is not defined externally. Instead, the linker then replaces the symbol with zero or some appropriate value. For example, if the label is not defined in the project, the linker then replaces it with the address of the next instruction after the branch.

```
IMPORT label [WEAK]
...
B label
...
```

Example 10-17. A symbol declared with weak prevents the linker from fatal linking error.

10.5.2 Example of Accessing Assembly Data in a C Program

To access variable *counter* defined in the assembly code, the C program has to use "extern int counter". This statement is to inform the compiler that this variable is defined outside this C program.

The assembly program allocates the memory space for the counter variable, and the C program only needs to indicate the existence of this variable without performing any memory allocation. Without the "extern" keyword, compilers would allocate space again for variable *counter* defined in the C program, thus producing an error of duplicated variables at the link stage.

Assembly Program

```
        AREA main, CODE
        EXPORT __main
        IMPORT getValue
        IMPORT increment
        IMPORT setValue
        ALIGN
        ENTRY

__main  MOVS   r2, #0
        MOVS   r0, #1
        BL     setValue
        BL     increment
        BL     getValue
        MOV    r2, r0
stop    B      stop

        AREA myData, DATA
        EXPORT counter
counter DCD    0
        END
```

C Program

```
extern int counter;

int getValue() {
   return counter;
}

void increment() {
   counter++;
}

void setValue(int c) {
   counter = c;
}
```

Example 10-18. C functions access a variable defined in an assembly program.

10.6 Exercises

1. Translate the following C statement into assembly.

```
x[1].c += 100;
```

The following gives the definition the structure array `x[2]`. Assume register `r0` holds the starting address of this array.

```
__packed struct X {
   uint8_t a;
   int32_t b;
   int16_t c;
   int32_t d;
   uint8_t e;
} x[2];
```

2. Translate the following C statement into assembly.

```
x[1].c += 100;
```

The structure array is defined below. Assume register `r0` contains the starting address of this array.

```
struct Y {
   uint8_t a;
   int32_t b;
   int16_t c;
   int32_t d;
   uint8_t e;
} y[2];
```

3. Write a subroutine in assembly that removes all occurrences of a given character in a string. The subroutine takes two parameters: the string pointer, and the character to be deleted. Write a C code that calls this subroutine. The string is defined as global in the C code.

4. Suppose we have the following *strcat* function written in C, which concatenates the second string to the first string. Write an assembly program that calls the strcat function. These two strings are defined in the data area in the assembly code.

```
void strcat (char * dst, char * src) {
    while(*dst++);
    while(*dst++ = *src++);
}
```

5. Write a subroutine *swap* in assembly that swaps two strings, and a C program that calls the swap subroutine. These two strings are defined in the C program. (Hint: There is no need to swap all characters in the strings, and swapping the memory pointers in the assembly is sufficient.)

6. Write an assembly program that calls the following C subroutine that returns the memory address of the last occurrence of a given character in a string.

```
char * search (char * s, char c) {
    char *p = NULL;
    for(; *s; s++)
        if (*s == c)
            p = s;
    return p;
}
```

7. Suppose the following structure array is defined as global in a C program. Write an assembly program that iterates through the array and find the total scores.

```
struct Student_T {
    char c1;
    char c2;
    int score;
    char c3;
} students[10];
```

8. Write a subroutine *max4* in assembly to find the maximum value among four signed integers. These integers are passed to the subroutine via registers. Write a C program to test the *max4* subroutine.

9. Write an assembly subroutine that checks whether a given integer is a palindrome number. For example, 9, 11, 1234321, 141, 1221, and 120021 are palindrome numbers. Write a C program that calls the assembly subroutine. The input is an unsigned integer. The return is 1 if the number is a palindrome and 0 if not.

10. Identify and correct the errors in the following inline assembly program that calculates the sum of four integers.

```
__asm int sum4(int a, int b, int c, int d)
      // arguments stored in r0 - r3
      MOV r4, r0        ; r0 = 1st argument
      ADD r4, r4, r1    ; r1 = 2nd argument
      ADD r4, r4, r2    ; r2 = 3rd argument
      ADD r0, r4, r3    ; r3 = 4th argument, r0 = return
}
```

```
int main(void){
      int s = sum4(1, 2, 3, 4);
      while(1);
}
```

11. Identify and correct the errors in the following inline assembly codes that calculate the sum of four integers.

```
int sum4(int a, int b, int c, int d){
     int t;
     __asm {
         ADD t, r0, r1;
         ADD t, r2;
         ADD t, r3;
     }
     return t;
}

int main(void){
     int s = sum4(1, 2, 3, 4);
     while(1);
}
```

12. Identify and correct the errors in the following two programs.

C Program (*main.c*)

```
char src[9] = "Hello!";
char dst[9];


int main(void){
    strcpy(dst, src);
    while(1);
}
```

Assembly Program (*strcpy.s*)

```
        AREA stringCopy, CODE
        ALIGN
strcpy  PROC
loop    LDRB r2, [r1] ; Load a byte, r1 = *src
        STRB r2, [r0] ; Store a byte, r0 = *dst
        ADD  r1, #1   ; Increase memory pointer
        ADD  r0, #1   ; Increase memory pointer
        CMP  r2, #0   ; Zero terminator
        BNE  loop     ; Loop if not null terminator
        ENDP
        END
```