Embedded Software in C for an ARM Cortex M

Jonathan W. Valvano and Ramesh Yerraballi (1/2015)

Chapter 0 The Preface

Zero is an appropriate place for a book on C to start. Zero has many special meanings to the C programmer. On the ARM Cortex M, zero is the address of the initial stack pointer that gets set on reset. The compiler will initialize all global variables to zero on start-up. We use a zero to signify the end of a string. A pointer with a zero value is considered a null-pointer (doesn't point to anything). We use a zero value to signify the Boolean false, and true is any nonzero value. The array subscripts in C start with zero.

This document serves as an introduction to C programming on the Texas Instruments TM4C123 or TM4C1294 microcomputers. Its purpose is to provide a short introduction to C programming in the context of Embedded Systems. Initially when we use general (not Embedded System specific) C constructs. we will assume you have access to a free compiler (codepad.org is an excellent start). For Embedded System development where a microcontroller board is used an IDE like the Keil uVision from ARM (**https://www.keil.com/demo/eval/armv4.htm**) is an invaluable tool to edit, compile and debug your code in both simulated enviroments as well as actual boards.

This document differs from classical C programming books in its emphasis on embedded systems. While reviewing the existing literature on C programming I was stuck by the high percentage of programming examples in these books that rely on the functions **scanf** and **printf** to perform input/output. While I/O is extremely important for embedded systems, rarely is serial I/O with **scanf** and **printf** an important aspect of an embedded system. This HTML document is clearly not comprehensive rather it serves as a short refresher for those C programmers whose skills are a little rusty. This document also assists the experienced programmer trained in another language like Pascal or C++, that now wishes to program in C for an embedded Cortex M system. If the reader in interested in a more classical approach to C programming I suggest:

A Book on C: Programming in C, by Kelley and Pohl, Addison-Wesley

Send comments and suggestions about this document to: *valvano@mail.utexas.edu or ramesh@mail.utexas.edu*

The style and structure of this HTML document was derived from A Small C Compiler: Language, Usage, Theory, and Design, by James E. Hendrix.

Legal Statement

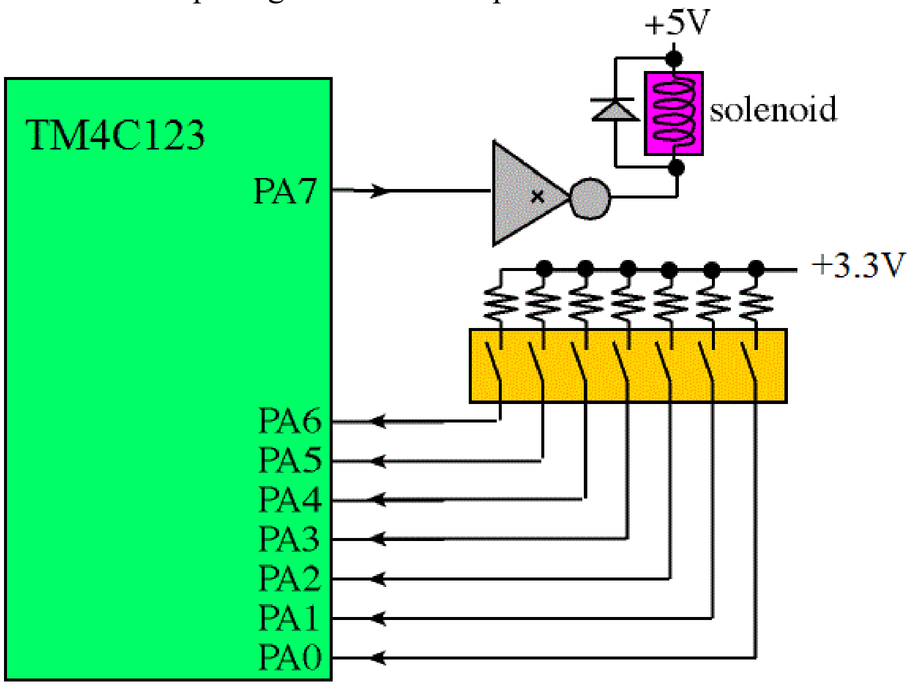
Chapter 1: Program Structure

What's in Chapter 1?

This chapter gives a basic overview of programming in C for an embedded system. We will introduce some basic terms so that you get a basic feel for the language. Since this is just the first of many chapters it is not important yet that you understand fully the example programs. The examples are included to illustrate particular features of the language.

Case Study : Microcomputer-Based Lock

To illustrate the software development process, we will implement a simple digital lock. The lock system has 7 toggle switches and a solenoid as shown in the following figure. If the 7-bit binary pattern on Port A bit 6 to bit 0 becomes 0100011 for at least 10 ms, then the solenoid will activate. The 10 ms delay will compensate for the switch bounce. For information on switches and solenoids see Sections 4.2 and 8.6.3 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano. For now what we need to understand is that Port A bits 6-0 are input signals to the computer and Port A bit 7 is an output signal.



Before we write C code, we need to develop a software plan. Software development is an iterative process. Even though we list steps the development process in a 1,2,3... order, in reality we iterative these steps over and over.

1) We begin with a list of the inputs and outputs. We specify the range of values and their significance. In this example we will use PORTA. Bits 6-0 will be inputs. The 7 input signals

represent an unsigned integer from 0 to 127. Port A bit 7 will be an output. If PA7 is 1 then the solenoid will activate and the door will be unlocked. In C, we use #define MACROS to assign a symbolic names to the corresponding addresses of the ports.

```
#define GPIO_PORTA_DATA_R       (*((volatile unsigned long *)0x400043FC))
#define GPIO_PORTA_DIR_R        (*((volatile unsigned long *)0x40004400))
#define GPIO_PORTA_DEN_R        (*((volatile unsigned long *)0x4000451C))
#define SYSCTL_PRGPIO_R         (*((volatile unsigned long *)0x400FEA08))
```

2) Next, we make a list of the required data structures. Data structures are used to save information. If the data needs to be permanent, then it is allocates in global space. If the software will change its value then it will be allocated in RAM. In this example we need a 32-bit unsigned counter.

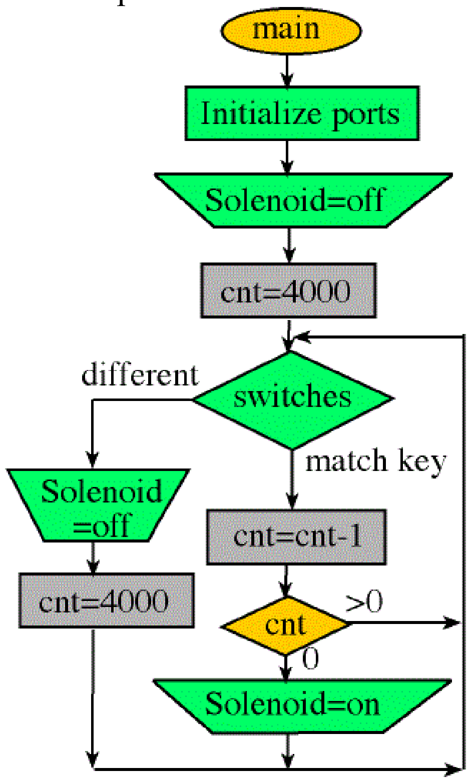
```
unsigned int cnt;
```

If data structure can be defined at compile time and will remain fixed, then it can be allocated in ROM. In this example we will define an 8-bit fixed constant to hold the key code, which the operator needs to set to unlock the door. The compiler will place these lines with the program so that they will be defined in Flash ROM memory.

```
const unsigned char key=0x23; // The key code 0100011 (binary)
```

It is not real clear at this point exactly where in ROM this constant will be, but luckily for us, the compiler will calculate the exact address automatically. After the program is compiled, we can look in the listing file or in the map file to see where in memory each structure is allocated.

3) Next we develop the software algorithm, which is a sequence of operations we wish to execute. There are many approaches to describing the plan. Experienced programmers can develop the algorithm directly in C language. On the other hand, most of us need an abstractive method to document the desired sequence of actions. Flowcharts and pseudo code are two common descriptive formats. There are no formal rules regarding pseudo code, rather it is a shorthand for describing what to do and when to do it. We can place our pseudo code as documentation into the comment fields of our program. The following shows a flowchart on the left and pseudo code and C code on the right for our digital lock example.



Pseudo Code

```
1) initialize ports
     PA6-PA0 inputs
     PA7 output
2) turn off solenoid
3) set counter to 4000
4) repeat indefinitely
   if switch matches key
     a) decrement counter
     b) if counter is zero
          turn on solenoid
   otherwise
     a) turn off solenoid
     b) set counter to 4000
```

C Code

```
PORTA=0; cnt=4000;
while(1){
  if((PORTA&0x7F==key){
    if((--cnt)==0)
      PORTA|=0x80;}
  else{
    PORTA=0; cnt=4000;}}
```

Normally we place the programs in Flash ROM. Typically, the compiler will initialize the stack pointer to the last location of RAM. On the ARM Cortex M, the stack is initialized to the 32-bit value located at ROM address 0. Next we write C code to implement the algorithm as illustrated in the above flowchart and pseudo code.

4) The last stage is debugging. For information on debugging see Sections 2.6, 4.6, 5.8 and 6.9 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

Free field language

In most programming languages the column position and line number affect the meaning. On the contrary, C is a free field language. Except for preprocessor lines (that begin with #, see Chapter 11), spaces, tabs and line breaks have the same meaning. The other situation where spaces, tabs and line breaks matter is string constants. We can not type tabs or line breaks within a string constant. For more information see the section on strings in the constants chapter. This means we can place more than one statement on a single line, or place a single statement across multiple lines. For example a function could have been written without any line breaks

```
void Lock_Init(void){ volatile unsigned long delay; SYSCTL_PRGPIO_R |=
0x01; delay = SYSCTL_PRGPIO_R;   GPIO_PORTA_DIR_R = 0x80;  GPIO_PORTA_DEN_R =
0xFF; }
```

"Since we rarely make hardcopy printouts of our software, it is not necessary to minimize the number of line breaks."

However, we could have added extra line breaks and comments to make it more readable.

```
void Lock_Init(void){ volatile unsigned long delay;
  SYSCTL_PRGPIO_R |= 0x01;   // activate clock for Port A
  delay = SYSCTL_PRGPIO_R;   // allow time for clock to start
  GPIO_PORTA_DIR_R = 0x80;   // set PA7 to output and PA6-0 to input
  GPIO_PORTA_DEN_R = 0xFF;   // enable digital port
}
```

At this point I will warn the reader, just because C allows many different forms of syntax, proper syntax will have a profound impact on the quality of our code. After much experience you will develop a programming style that is easy to understand. Although spaces, tabs, and line breaks are syntactically equivalent, their proper usage will have a profound impact on the readability of your software. For more information on programming style see Section 5.6 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

A *token* in C can be a user defined name (e.g., the variable **`Info`** and function **Lock_Init**) or a predefined operation (e.g., **`*, unsigned, while`**). Each token must be contained on a single line. We see in the above example that tokens can be separated by white spaces (space, tab, line break) or by the special characters, which we can subdivide into punctuation marks (Table 1-1) and operations (Table 1-2). Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both the beginning and experienced programmers.

punctuation	Meaning
;	End of statement
:	Defines a label
,	Separates elements of a list
( )	Start and end of a parameter list
{ }	Start and stop of a compound statement
[ ]	Start and stop of a array index

" "	Start and stop of a string
' '	Start and stop of a character constant

Table 1-1: Special characters can be punctuation marks

The next table shows the single character operators. For a description of these operations, see Chapter 5.

operation	Meaning
=	assignment statement
@	address of
?	selection
<	less than
>	greater than
!	logical not (true to false, false to true)
~	1's complement
+	addition
-	subtraction
*	multiply or pointer reference
/	divide
%	modulo, division remainder
|	logical or
&	logical and, or address of
^	logical exclusive or
.	used to access parts of a structure

Table 1-2: Special characters can be operators

The next table shows the operators formed with multiple characters. For a description of these operations, see Chapter 5.

operation	Meaning
==	equal to comparison
<=	less than or equal to
>=	greater than or equal to
!=	not equal to
<<	shift left
>>	shift right
++	increment
--	decrement
&&	Boolean and
||	Boolean or
+=	add value to
-=	subtract value to
*=	multiply value to
/=	divide value to
|=	or value to
&=	and value to
^=	exclusive or value to
<<=	shift value left
>>=	

	shift value right
`%=`	modulo divide value to
`->`	pointer to a structure

Table 1-3: Multiple special characters also can be operators

Although the operators will be covered in detail in Chapter 9, the following section illustrates some of the common operators. We begin with the assignment operator. Notice that in the line `x=1;` x is on the left hand side of the = . This specifies the address of x is the destination of assignment. On the other hand, in the line `z=x;` x is on the right hand side of the = . This specifies the value of x will be assigned into the variable z. Also remember that the line `z=x;` creates two copies of the data. The original value remains in x, while z also contains this value.

```
short x,y,z; /* Three variables */
void Example(void){
   x = 1;          /* set the value of x to 1 */
   y = 2;          /* set the value of y to 2 */
   z = x;          /* set the value of z to the value of x (both are 1) */
   x = y = z = 0;  /* all all three to zero   */
}
```

Listing 1-2: Simple program illustrating C arithmetic operators

Next we will introduce the arithmetic operations addition, subtraction, multiplication and division. The standard arithmetic precedence apply. For a detailed description of these operations, see Chapter 5.

```
short x,y,z; /* Three variables */
void Example(void){
   x=1; y=2;   /* set the values of x and y */
   z = x+4*y;  /* arithmetic operation */
   x++;        /* same as x=x+1;   */
   y--;        /* same as y=y-1;   */
   x = y<<2;   /* left shift same as x=4*y;   */
   z = y>>2;   /* right shift same as x=y/4;  */
   y += 2;     /* same as y=y+2;   */
}
```

Listing 1-3: Simple program illustrating C arithmetic operators

Next we will introduce a simple conditional control structure. Assume PORTB is configured as an output port, and PORTE as an input port. For more information on input/output ports see Chapter 4 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano. The expression `PORTE&0x04` will return 0 if PORTE bit 2 is 0 and will return a 4 if PORTE bit 2 is 1. The expression `(PORTE&0x04)==0` will return TRUE if PORTE bit 2 is 0 and will return a FALSE if PORTE bit 2 is 1. The statement immediately following the `if` will be executed if the condition is TRUE. The `else` statement is optional.

```
#define PORTB (*((volatile unsigned long *)0x400053FC))
#define PORTE (*((volatile unsigned long *)0x400243FC))
void Example(void){
  if((PORTE&0x04)==0){ /* test bit 2 of PORTE */
    PORTB = 0;       /* if PORTE bit 2 is 0, then make PORTB=0 */
  }else{
    PORTB = 100;     /* if PORTE bit 0 is not 0, then make PORTB=100 */
  }
}
```

Listing 1.4: Simple program illustrating the C if else control structure

In the next example lets assume that PORTA bit 3 is configured as an output pin on the TM4C123. Like the `if` statement, the `while` statement has a conditional test (i.e., returns a TRUE/FALSE). The statement immediately following the `while` will be executed over and over until the conditional test becomes FALSE.

```
#define PORTA (*((volatile unsigned long *)0x400043FC))
#define PORTB (*((volatile unsigned long *)0x400053FC))
void Example(void){ /* loop until PORTB equals 200 */
  PORTB = 0;
  while(PORTB != 200){
    PORTA = PORTA^0x08;}  /* toggle PORTA bit 3 output */
    PORTB++;}             /* increment PORTB output */
}
```

Listing 1.5: Simple program illustrating the C while control structure

The `for` control structure has three parts and a body. `for(part1;part2;part3){body;}` The first part `PORTB=0` is executed once at the beginning. Then the body `PORTA = PORTA^0x08;` is executed, followed by the third part `PORTB++`. The second part `PORTB!=200` is a conditional. The body and third part are repeated until the conditional is FALSE. For a more detailed description of the control structures, see Chapter 6.

```
#define PORTB (*((volatile unsigned long *)0x400053FC))
void Example(void){       /* loop until PORTB equals 200 */
  for(PORTB=0; PORTB != 200; PORTB++){
    PORTA = PORTA^0x08;}  /* toggle PORTA bit 3 output */
  }
}
```

Listing 1.6: Simple program illustrating the C for loop control structure

Precedence

As with all programming languages the order of the tokens is important. There are two issues to consider when evaluating complex statements. The **precedence** of the operator determines which operations are performed first. In the following example, the 2*x is performed first because * has higher precedence than + and =. The addition is performed second because + has higher precedence than =. The assignment = is performed last. Sometimes we use parentheses to clarify the meaning of the expression, even when they are not needed. Therefore, the line **z=y+2*x;** could also have been written **z=2*x+y;** or **z=y+(2*x);** or **z=(2*x)+y;**.

```
short example(short x, short y){ short z;
  z = y+2*x;
  return(z);
}
```

The second issue is the **associativity**. Associativity determines the left to right or right to left order of evaluation when multiple operations of the precedence are combined. For example + and - have the same precedence, so how do we evaluate the following?

```
z = y-2+x;
```

We know that + and - associate the left to right, this function is the same as **z=(y-2)+x;**. Meaning the subtraction is performed first because it is more to the left than the addition. Most operations associate left to right, but the following table illustrates that some operators associate right to left.

Precedence	Operators	Associativity
highest	() [] . -> ++(postfix) --(postfix)	left to right
	++(prefix) --(prefix) !~ **sizeof**(type) +(unary) -(unary) &(address) *(dereference)	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= <<= >>= |= &= ^=	right to left
lowest	,	left to right

Table 1-4: Precedence and associativity determine the order of operation

"When confused about precedence (and aren't we all) add parentheses to clarify the expression."

Comments

There are two types of comments. The first type explains how to use the software. These comments are usually placed at the top of the file, within the header file, or at the start of a function. The reader of these comments will be writing software that uses or calls these routines. The second type of comments assists a future programmer (ourselves included) in changing, debugging or extending these routines. We usually place these comments within the body of the functions. The comments on the right of each line are examples of the second type. For more information on writing good comments see Section 5.6 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

Comments begin with the `/*` sequence and end with the `*/` sequence. They may extend over multiple lines as well as exist in the middle of statements. The following is the same as `PORTA=0x80;`

```
PORTA /*PA7 is output*/=0x80/*turn on relay*/;
```

Keil uVision does allow the use of C++ style comments. The start comment sequence is `//` and the comment ends at the next line break or end of file. Thus, the following two lines are equivalent:

```
UART_Init(); /* turn on UART serial port */
UART_Init(); // turn on UART serial port
```

C does allow the comment start and stop sequences within character constants and string constants. For example the following string contains all 7 characters, not just the `ac`

```
const char str[10]="a/*b*/c";
```

Most compilers unfortunately do not support comment nesting. This makes it difficult to comment out sections of logic that are themselves commented. For example, the following attempt to comment-out the `PORTA = 0x00;` will result in a compiler error.

```
int main(void){ unsigned char data;
  Lock_Init();      /* initialize I/O ports */
/*
  PORTA = 0x00;     /* output to port A */
*/
  while(1){
    Info = PORTE;     /* input from port E */
    PORTB = Info;}}   /* output to port B */
```

The conditional compilation feature can be used to temporarily remove and restore blocks of code.

Preprocessor Directives

Preprocessor directives begin with `#` in the first column. As the name implies preprocessor commands are processed first. I.e., the compiler passes through the program handling the preprocessor directives. Although there are many possibilities (assembly language, conditional compilation, interrupt service routines), I thought I'd mention the two most important ones early in this document. We have already seen the macro definition (`#define`) used to define I/O ports and bit fields. A second important directive is the `#include`, which allows you to include another entire file at that position within the program. The following directive will define all the TM4C123 I/O port names.

```
#include "tm4c123gh6pm.h"
```

Examples of `#include` are shown below, and more in Chapter 11.

Global Declarations

An object may be a data structure or a function. Objects that are not defined within functions are global. Objects that may be declared in Keil uVision include:

integer variables (8-bit 16-bit or 32-bit signed or unsigned)
character variables (8-bit)
arrays of integers or characters
pointers to integers or characters
arrays of pointers
structure (grouping of other objects)
unions (redefinitions of storage)
functions

Keil uVision supports 64-bit **long long** integers and floating point. In this document we will focus on 8-bit 16-bit and 32-bit objects. Object code generated with most ARM compilers is often more efficient using 32-bit parameters rather than 8-bit or 16-bit parameters.

Declarations and Definitions

It is important for the C programmer to distinguish the two terms *declaration* and *definition*. A function declaration specifies its name, its input parameters and its output parameter. Another name for a function declaration is *prototype*. A data structure declaration specifies its type and format. On the other hand, a function definition specifies the exact sequence of operations to execute when it is called. A function definition will generate object code (machine instructions to be loaded into memory that perform the intended operations). A data structure definition will reserve space in memory for it. The confusing part is that the definition will repeat the declaration specifications. We can declare something without defining it, but we cannot define it without declaring it. For example the declaration for the function `UART_OutChar` could be written as

```
void UART_OutChar(char);

or

void UART_OutChar(char letter);
```

We can see that the declaration shows us how to use the function, not how the function works. Because the C compilation is a one-pass process, an object must be declared or defined before it can be used in a statement. (Actually the preprocess performs a pass through the program that handles the preprocessor directives.) One approach is to first declare a functions, use it, and lastly defines the functions:

```
long add (long, long);

int main(void) {
  long x,y,z;
  x = 2; y = 3;
  z = add(x,y);
  return 1;
}

long add(long a, long b){
  return (a+b);
}
```

An object may be said to exist in the file in which it is defined, since compiling the file yields a module containing the object. On the other hand, an object may be declared within a file in which it does not exist. Declarations of data structures are preceded by the keyword **extern**. Thus,

```
long RunFlag;
```

defines 32-bit signed integer called `RunFlag`; whereas,

```
extern long RunFlag;
```

only declares the **`RunFlag`** to exist in another, separately compiled, module. Thus the line

```
extern void SysTick_Handler();
```

declares the function name and type just like a regular function declaration. The **extern** tells the compiler that the actual function exists in another module and the linker will combine the modules so that the proper action occurs at run time. The compiler knows everything about extern objects except where they are. The linker is responsible for resolving that discrepancy. The compiler simply tells the assembler that the objects are in fact external. And the assembler, in turn, makes this known to the linker.

Functions

A *function* is a sequence of operations that can be invoked from other places within the software. We can pass 0 or more parameters into a function. The code generated by the Keil uVision C compiler passes the first four input parameters in Register R0, R1, R2 and R3 and the remaining parameters are passed on the stack. A function can have 0 or 1 output parameter. The return parameter is placed in Register R0 (8-bit or 16-bit return parameters are promoted to 32 bits.) The `add` function below (an improvement that checks for overflow) has two 16-bit signed input parameters, and one 16-bit output parameter. Again the numbers in the first column are not part of the software, but added to simplify our discussion.

```
1   short add(short x, short y){ short z;
2     z = x+y;
3     if((x>0)&&(y>0)&&(z<0))z=32767;
4     if((x<0)&&(y<0)&&(z>0))z=-32768;
5     return(z);}
6   int main(void){ short a,b;
7     a = add(2000,2000)
8     b = 0
9     while(1){
10       b = add(b,1);
11   }
```

Listing 1-8: Example of a function call

The interesting part is that after the operations within the function are performed, control returns to the place right after where the function was called. In C, execution begins with the **main** program. The execution sequence is shown below:

```
6   int main(void){ short a,b;
7     a = add(2000,2000);            /* call to add*/
1   short add(short x, short y){ short z;
2     z = x+y;                       /* z=4000*/
3     if((x>0)&&(y>0)&&(z<0))z=32767;
4     if((x<0)&&(y<0)&&(z>0))z=-32768;
5     return(z);}             /* return 4000 from call*/
8     b = 0
9     while(1){
10       b = add(b,1); }            /* call to add*/
1   short add(short x, short y){ short z;
2     z = x+y;                       /* z=1*/
3     if((x>0)&&(y>0)&&(z<0))z=32767;
4     if((x<0)&&(y<0)&&(z>0))z=-32768;
5     return(z);}             /* return 1 from call*/
11   }
9     while(1){
10       b = add(b,1); }            /* call to add*/
1   short add(short x, short y){ short z;
2     z = x+y;                       /* z=2*/
3     if((x>0)&&(y>0)&&(z<0))z=32767;
4     if((x<0)&&(y<0)&&(z>0))z=-32768;
5     return(z);}             /* return 2 from call*/
11   }
```

Notice that the return from the first call goes to line 8, while all the other returns go to line 11. The execution sequence repeats lines 9,10,1,2,3,4,5,11 indefinitely.

The programming language Pascal distinguishes between functions and procedures. In Pascal a function returns a parameter while a procedure does not. C eliminates the distinction by accepting a bare or `void` expression as its return parameter.

C does not allow for the nesting of procedural declarations. In other words you can not define a function within another function. In particular all function declarations must occur at the global level.

A function declaration consists of two parts: a *declarator* and a *body*. The declarator states the name of the function and the names of arguments passed to it. The names of the argument are only used inside the function. In the add function above, the declarator is **(short x, short y)** meaning it has two 16-bit input parameters.

The parentheses are required even when there are no arguments. When there are no input parameters a `void` or nothing can be specified. The following two statements are equivalent:

```
void TogglePA3(void){PORTA ^= 0x08;}
void TogglePA3(){PORTA ^= 0x08;}
```

I prefer to include the `void` because it is a positive statement that there are no input parameters. However, one must specify the output parameter, even if there is none. For more information on functions see Chapter 10.

The body of a function consists of a statement that performs the work. Normally the body is a compound statement between a {} pair. If the function has a return parameter, then all exit points must specify what to return. In the following median filter function shown in Listing 1-4, there are six possible exit paths that all specify a return parameter.

On a reset or a power initialization, the 32-bit value in ROM location 0x000000004 is loaded into the program counter, PC. The programs created using Keil uVision actually begin execution at a place called **Reset_Handler**, which can be found in the **start.s** file. After a power on or hardware reset, the embedded system will initialize the stack, initialize the heap, and clear all RAM-based global variables. After this brief initialization sequence the function named **main()** is called. Consequently, there must be a **main()** function somewhere in the program. If you are curious about what really happens, look in the assembly file **start.s** For programs not in an embedded environment (e.g., running on your PC) a return from **main()** transfers control back to the operating system. As we saw earlier, software for an embedded system usually does not quit.

Compound Statements

A *compound statement* (or *block*) is a sequence of statements, enclosed by braces, that stands in place of a single statement. Simple and compound statements are completely interchangeable as far as the syntax of the C language is concerned. Therefore, the statements that comprise a compound statement may themselves be compound; that is, blocks can be nested. Thus, it is legal to write

```
// 3 wide 16-bit signed median filter
short median(short n1,short n2,short n3){
    if(n1>n2){
      if(n2>n3)
        return(n2);    // n1>n2,n2>n3    n1>n2>n3
      else{
      if(n1>n3)
        return(n3);    // n1>n2,n3>n2,n1>n3 n1>n3>n2
      else
        return(n1);    // n1>n2,n3>n2,n3>n1 n3>n1>n2
      }
    }
    else{
      if(n3>n2)
        return(n2);    // n2>n1,n3>n2     n3>n2>n1
      else{
        if(n1>n3)
          return(n1);  // n2>n1,n2>n3,n1>n3 n2>n1>n3
        else
          return(n3);  // n2>n1,n2>n3,n3>n1 n2>n3>n1
      }
    }
}
```

Listing 1-9: Example of nested compound statements.

Although C is a free-field language, notice how the indenting has been added to the above example. The purpose of this indenting is to make the program easier to read. On the other hand since C is a free-field language, the following two statements are quite different

```
if(n1>100) n2=100; n3=0;
if(n1>100) {n2=100; n3=0;}
```

In both cases `n2=100;` is executed if `n1>100`. In the first case the statement `n3=0;` is always executed, while in the second case `n3=0;` is executed only if `n1>100`.

Global Variables

Variables declared outside of a function, like `Count` in the following example, are properly called *external* variables because they are defined outside of any function. While this is the standard term for these variables, it is confusing because there is another class of external variable, one that exists in a separately compiled source file. In this document we will refer to variables in the present source file as *globals*, and we will refer to variables defined in another file as externals.

There are two reasons to employ global variables. The first reason is data permanence. The other reason is information sharing. Normally we pass information from one module to another explicitly using input and output parameters, but there are applications like interrupt programming where this method is unavailable. For these situations, one module can store data into a global while another module can view it.

In the following example, we wish to maintain a counter of the number of times PA3 is toggled. This data must exist for the entire life of the program. This example also illustrates that with an embedded system it is important to initialize RAM-based globals at run time. Most C compilers (including uVision) will automatically initialize globals to zero at startup.

```
unsigned long Count;  /* number of toggles, initialized to 0 */
void TogglePA3(void){
  Count = Count+1; /* incremented each time called */
  PORTA ^= 0x08;}
```

Listing 1-10: A global variable contains permanent information

Although the following two examples are equivalent, I like the second case because its operation is more self-evident. In both cases the global is allocated in RAM, and initialized at the start of the program to 1.

```
int Flag = 1;
void main(void) {
/* main body goes here */
}
```

Listing 1-11: A global variable initialized at run time by the compiler

```
int Flag;
void main(void) { Flag=1;
/* main body goes here */
}
```

Listing 1-12: A global variable initialized at run time by the compiler

From a programmer's point of view, we usually treat the I/O ports in the same category as global variables because they exist permanently and support shared access.

Local Variables

Local variables are very important in C programming. They contain temporary information that is accessible only within a narrow scope. We can define local variables at the start of a compound statement. We call these *local variables* since they are known only to the block in which they appear, and to subordinate blocks. The following statement adjusts x and y such that x contains the smaller number and y contains the larger one. If a swap is required then the local variable z is used.

```
if(x>y){ short z;    /* create a temporary variable */
  z=x; x=y; y=z;     /* swap x and y */
}                    /* then destroy z */
```

Notice that the local variable z is declared within the compound statement. Unlike globals, which are said to be *static*, locals are created dynamically when their block is entered, and they cease to exist when control leaves the block. Furthermore, local names supersede the names of globals and other locals declared at higher levels of nesting. Therefore, locals may be used freely without regard to the names of other variables. Although two global variables can not use the same name, a local variable of one block can use the same name as a local variable in another block. Programming errors and confusion can be avoided by understanding these conventions.

Source Files

Our programs may consist of source code located in more than one file. The simplest method of combining the parts together is to use the #include preprocessor directive. Another method is to compile the source files separately, then combine the separate object files as the program is being linked with library modules. The linker/library method should be used when the programs are large, and only small pieces are changed at a time. On the other hand, most embedded system applications are small enough to use the simple method. In this way we will compile the entire system whenever changes are made. Remember that a function or variable must be defined or declared before it can be used. The following example is one method of dividing our simple example into multiple files.

```
/* ****file tm4c123gh6pm.h (actually much bigger)************ */
#define GPIO_PORTA_DATA_R   (*((volatile unsigned long *)0x400043FC))
#define GPIO_PORTA_DIR_R    (*((volatile unsigned long *)0x40004400))
#define GPIO_PORTA_DEN_R    (*((volatile unsigned long *)0x4000451C))
#define SYSCTL_PRGPIO_R     (*((volatile unsigned long *)0x400FEA08))
```

Listing 1-13: Header file for Port A I/O ports

```
/* ****file LOCK.h ************ */
void Lock_Init(void);
void Lock_Set(int flag);
unsigned long Lock_Input(void);
```

Listing 1-14: Header file for the Port A functions

```
/* ****file Lock.C ************ */
#include "tm4c123gh6pm.h"
void Lock_Init(void){ volatile unsigned long delay;
  SYSCTL_PRGPIO_R |= 0x01;   // activate clock for Port A
  delay = SYSCTL_PRGPIO_R;   // allow time for clock to start
  GPIO_PORTA_DIR_R = 0x80;   // set PA7 to output and PA6-0 to input
  GPIO_PORTA_DEN_R = 0xFF;   // enable digital port
}
void Lock_Set(int flag){
  if(flag){
    GPIO_PORTA_DATA_R = 0x80;
  }else{
    GPIO_PORTA_DATA_R = 0;
  }
}
unsigned long Lock_Input(void){
  return GPIO_PORTA_DATA_R&0x7F; // 0 to 127
}
```

Listing 1-15: Implementation file for the Port A interface

/* ****file main.c ************ */

```
const unsigned char key=0x23; // The key code 0100011 (binary)
#include "Lock.h"
void main(void){ unsigned char input; unsigned long cnt;
  Lock_Init(); // initialize lock
  cnt = 4000;
  while(1){
    key = Lock_Input(); // input 8 bits from parallel port A
    if(key == input){
      cnt--;         // debounce switches
      if(cnt == 0){  // done bouncing
        Lock_Set(1); // unlock door
      }
    }else{
      Lock_Set(0);   // lock the door
      cnt = 4000;
    }
  }
}
#include "Lock.c"
```

Listing 1-16: Main program file for this system

With Keil uVision, we do not need the `#include "Lock.c"` because Lock.c will be included in the project. I make the following general statement about good programming style.

"If the software is easy to understand, debug, and change, then it is written with good style"

While the main focus of this document is on C syntax, it would be improper to neglect all style issues. This system was divided using the following principles:

Define the I/O ports in a **tm4c123gh6pm.h** header file
For each module place the user-callable prototypes in a *.h header file
For each module place the implementations in a *.c program file
In the main program file, include the header files first
In the main program file, include the implementation files last

Breaking a software system into files has a lot of advantages. The first reason is code reuse. Consider the code in this example. If a Lock output function is needed in another application, then it would be a simple matter to reuse the **lock.h** and **lock.c** files. The next advantage is clarity. Because the details have been removed, the overall approach is easier to understand. The next reason to break software into files is parallel development. As the software system grows it will be easier to divide up a software project into subtasks, and to recombine the modules into a complete system if the subtasks have separate files. The last reason is upgrades. Consider an upgrade in our simple example where the Port A is replaced with Port B. For this kind of upgrade we implement the Port B functions in the **Lock.c** file with the new version. If we plan appropriately, we should be able to make this upgrade without changes to the files **lock.h** and **main.c**.

Go to Chapter 2 on Tokens Return to Table of Contents

Chapter 2: Tokens

What's in Chapter 2?

This chapter defines the basic building blocks of a C program. Understanding the concepts in this chapter will help eliminate the syntax bugs that confuse even the veteran C programmer. A simple syntax error can generate 100's of obscure compiler errors. In this chapter we will introduce some of the syntax of the language.

To understand the syntax of a C program, we divide it into *tokens* separated by *white spaces* and *punctuation*. Remember the white spaces include space, tab, carriage returns and line feeds. A token may be a single character or a sequence of characters that form a single item. The first step of a compiler is to process the program into a list of tokens and punctuation marks. The following example includes punctuation marks of `( ) { } ;` The compiler then checks for proper syntax. And, finally, it creates object code that performs the intended operations. In the following example:

```
void main(void){ long z;
  z = 0;
  while(1){
    z = z+1;
  }
}
```

Listing 2-1: Example of a function call

The following sequence shows the tokens and punctuation marks from the above listing:

```
void main ( void ) { long z ; z = 0 ; while ( 1 ) { z = z + 1 ; } }
```

Since tokens are the building blocks of programs, we begin our study of C language by defining its tokens.

ASCII Character Set

Like most programming languages C uses the standard ASCII character set. The following table shows the 128 standard ASCII code. One or more *white space* can be used to separate tokens and or punctuation marks. The white space characters in C include horizontal tab (9=0x09), the carriage return (13=0x0D), the line feed (10=0x0A), space (32=0x20).

BITS 4 to 6

		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
B	1	SOH	DC1	!	1	A	Q	a	q
I	2	STX	DC2	"	2	B	R	b	r
T	3	ETX	DC3	#	3	C	S	c	s

S	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
0	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
3	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	S1	US	/	?	O	_	o	DEL

Table 2-1. ASCII Character codes.

The first 32 (values 0 to 31 or 0x00 to 0x1F) and the last one (127=0x7F) are classified as *control characters*. Codes 32 to 126 (or 0x20 to 0x7E) include the "normal" characters. Normal characters are divided into

the space character (32=0x20),
the numeric digits 0 to 9 (48 to 57 or 0x30 to 0x39),
the uppercase alphabet A to Z (65 to 90 or 0x41 to 0x5A),
the lowercase alphabet a to z (97 to122 or 0x61 to 0x7A), and
the special characters (all the rest).

Literals

Numeric literals consist of an uninterrupted sequence of digits delimited by white spaces or special characters (operators or punctuation). Although Metrowerks does support floating point, this document will not cover it. The use of floating point requires a substantial about of program memory and execution time, therefore most applications should be implemented using integer math. Consequently the period will not appear in numbers as described in this document. For more information about numbers see the sections on decimals, octals, or hexadecimals in Chapter 3.

Character literals are written by enclosing an ASCII character in apostrophes (single quotes). We would write `'a'` for a character with the ASCII value of the lowercase a (97). The control characters can also be defined as constants. For example `'\t'` is the tab character. For more information about character literals see the section on characters in Chapter 3.

String literals are written as a sequence of ASCII characters bounded by quotation marks (double quotes). Thus, "ABC" describes a string of characters containing the first three letters of the alphabet in uppercase. For more information about string literals see the section on strings in Chapter 3.

Keywords

There are some predefined tokens, called *keywords*, that have specific meaning in C programs. The reserved words we will cover in this document are:

keyword	*meaning*
auto	Specifies a variable as automatic (created on the stack)
break	Causes the program control structure to finish
case	One possibility within a switch statement

char	8-bit integer
const	Defines global parameter as constant in ROM, and defines a local parameter as fixed value
continue	Causes the program to go to beginning of loop
default	Used in switch statement for all other cases
do	Used for creating program loops
double	Specifies variable as double precision floating point
else	Alternative part of a conditional
extern	Defined in another module
float	Specifies variable as single precision floating point
for	Used for creating program loops
goto	Causes program to jump to specified location
if	Conditional control structure
int	32-bit integer (same as long on the ARM) It should be avoided in most cases because the implementation will vary from compiler to compiler.
long	32-bit integer
register	Specifies how to implement a local
return	Leave function
short	16-bit integer
signed	Specifies variable as signed (default)
sizeof	Built-in function returns the size of an object
static	Stored permanently in memory, accessed locally
struct	Used for creating data structures
switch	Complex conditional control structure
typedef	Used to create new data types
unsigned	Always greater than or equal to zero
void	Used in parameter list to mean no parameter
volatile	Can change implicitly outside the direct action of the software. It disables compiler optimization, forcing the compiler to fetch a new value each time
while	Used for creating program loops

Table 2-2. Keywords have predefined meanings.

Did you notice that all of the keywords in C are lowercase? Notice also that as a matter of style, I used a mixture of upper and lowercase for the names I created, and all uppercase for the I/O ports. It is a good programming practice not to use these keywords for your variable or function names.

Names

We use *names* to identify our variables, functions, and macros. Names must begin with a letter or underscore and the remaining characters must be either letters or digits. We can use a mixture of upper and lower case or the underscore character to create self-explaining symbols. E.g.,

```
time_of_day    go_left_then_stop

TimeOfDay      GoLeftThenStop
```

The careful selection of names goes a long way to making our programs more readable. Names may be written with both upper and lowercase letters. The names are case sensitive. Therefore the following names are different:

```
thetemperature
THETEMPERATURE
TheTemperature
```

The practice of naming macros in uppercase calls attention to the fact that they are not variable names but defined symbols. Remember the I/O port names are implemented as macros in the header file **tm4c123gh6pm.h**.

We can use the map file to get the absolute addresses for these labels, then use the debugger to observe and modify their contents.

Developing a naming convention will avoid confusion. Possible ideas to consider include:

1. Start every variable name with its type. E.g.,

b	means Boolean true/false
s8	means 8-bit signed integer
u8	means 8-bit unsigned integer
s16	means 16-bit signed integer
u16	means 16-bit unsigned integer
s32	means 32-bit signed integer
u32	means 32-bit unsigned integer
c	means 8-bit ASCII character
s	means null terminated ASCII string

2. Start every local variable with "the" or "my"

3. Start every global variable and function with associated file or module name. In the following example the names all begin with `Bit_`. Notice how similar this naming convention recreates the look and feel of the modularity achieved by classes in C++. E.g.,

```
/* **********file=Bit.c*************
   Pointer implementation of the a Bit_Fifo
   These routines can be used to save (Bit_Put) and
   recall (Bit_Get) binary data 1 bit at a time (bit streams)
   Information is saved/recalled in a first in first out manner
   Bit_FifoSize is the number of 16-bit words in the Bit_Fifo
   The Bit_Fifo is full when it has 16*Bit_FifoSize-1 bits */
#define Bit_FifoSize 4
// 16*4-1=31 bits of storage
unsigned short Bit_Fifo[Bit_FifoSize]; // storage for Bit Stream
struct Bit_Pointer{
   unsigned short Mask; // 0x8000, 0x4000,...,2,1
   unsigned short *WPt;}; // Pointer to word containing bit
typedef struct Bit_Pointer Bit_PointerType;
Bit_PointerType Bit_PutPt; // Pointer of where to put next
Bit_PointerType Bit_GetPt; // Pointer of where to get next
/* Bit_FIFO is empty if Bit_PutPt==Bit_GetPt */
/* Bit_FIFO is full if Bit_PutPt+1==Bit_GetPt */
short Bit_Same(Bit_PointerType p1, Bit_PointerType p2){
   if((p1.WPt==p2.WPt)&&(p1.Mask==p2.Mask))
      return(1); //yes
   return(0);} // no
void Bit_Init(void) {
   Bit_PutPt.Mask=Bit_GetPt.Mask=0x8000;
   Bit_PutPt.WPt=Bit_GetPt.WPt=&Bit_Fifo[0]; /* Empty */
}
// returns TRUE=1 if successful,
// FALSE=0 if full and data not saved
// input is boolean FALSE if data==0
short Bit_Put(short data) { Bit_PointerType myPutPt;
   myPutPt=Bit_PutPt;
   myPutPt.Mask=myPutPt.Mask>>1;
   if(myPutPt.Mask==0) {
      myPutPt.Mask=0x8000;
      if((++myPutPt.WPt)==&Bit_Fifo[Bit_FifoSize])
         myPutPt.WPt=&Bit_Fifo[0]; // wrap
   }
   if (Bit_Same(myPutPt,Bit_GetPt))
      return(0); /* Failed, Bit_Fifo was full */
   else {
      if(data)
         (*Bit_PutPt.WPt) |= Bit_PutPt.Mask; // set bit
      else
         (*Bit_PutPt.WPt) &= ~Bit_PutPt.Mask; // clear bit
      Bit_PutPt=myPutPt;
      return(1);
   }
}
// returns TRUE=1 if successful,
// FALSE=0 if empty and data not removed
// output is boolean 0 means FALSE, nonzero is true
short Bit_Get(unsigned short *datapt) {
   if (Bit_Same(Bit_PutPt,Bit_GetPt))
      return(0); /* Failed, Bit_Fifo was empty */
   else {
      *datapt=(*Bit_GetPt.WPt)&Bit_GetPt.Mask;
      Bit_GetPt.Mask=Bit_GetPt.Mask>>1;
      if(Bit_GetPt.Mask==0) {
         Bit_GetPt.Mask=0x8000;
         if((++Bit_GetPt.WPt)==&Bit_Fifo[Bit_FifoSize])
            Bit_GetPt.WPt=&Bit_Fifo[0]; // wrap
      }
      return(1);
   }
}
```

Listing 2-2: This naming convention can create modularity similar to classes in C++.

Punctuation

Punctuation marks (semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses) are very important in C. It is one of the most frequent sources of errors for both the beginning and experienced programmers.

Semicolons

Semicolons are used as statement terminators. Strange and confusing syntax errors may be generated when you forget a semicolon, so this is one of the first things to check when trying to remove syntax errors. In this example we assume that Port B has been initialized as an output. Notice that one semicolon is placed at the end of every simple statement in the following example

```
#define PORTB (*((volatile unsigned long *)0x400053FC))
void Step(void){
  PORTB = 10;
  PORTB = 9;
  PORTB = 5;
  PORTB = 6;
}
```

Listing 2-3: Semicolons are used to separate one statement from the next.

Preprocessor directives do not end with a semicolon since they are not actually part of the C language proper. Preprocessor directives begin in the first column with the #and conclude at the end of the line. The following example will fill the array `DataBuffer` with data read from the input port (PORTB). We assume in this example that Port B has been initialized as an input. Semicolons are also used in the `for loop` statement (see also Chapter 6), as illustrated by

```
void Fill(void){ short j;
  for(j=0; j<100; j++){
    DataBuffer[j] = PORTB;
  }
}
```

Listing 2-4: Semicolons are used to separate three fields of the for statement.

Colons

We can define a label using the colon. Although C has a `goto` statement, I strongly discourage its use. I believe the software is easier to understand using the block-structured control statements (`if`, `if else`, `for`, `while`, `do while`, and `switch case`.) The following example will return after the Port B input reads the same value 100 times in a row. In this example, we assume Port B has been initialized as an input. Notice that every time the current value on Port B is different from the previous value the counter is reinitialized.

```
char Debounce(void){ short Cnt; unsigned char LastData;
Start:    Cnt=0;          /* number of times Port C is the same */
          LastData=PORTB;
Loop:     if(++Cnt==100) goto Done;     /* same thing 100 times */
          if(LastData!=PORTB) goto Start;/* changed */
          goto Loop;
Done:     return(LastData);
}
```

Listing 2-4: Colons are used to define labels (places we can jump to)

Colons also terminate `case`, and `default` prefixes that appear in switch statements. For more information see the section on switch in Chapter 6. In the following example, the next stepper motor output is found (the proper sequence is 10,9,5,6). The default case is used to restart the pattern.

```
unsigned char NextStep(unsigned char step){ unsigned char theNext;
  switch(step){
    case 10: theNext=9; break;
    case 9: theNext=5; break;
    case 5: theNext=6; break;
    case 6: theNext=10; break;
    default: theNext=10;
  }
  return(theNext);
}
```

Listing 2-5: Colons are also used to with the switch statement

For both applications of the colon (`goto` and `switch`), we see that a label is created that is a potential target for a transfer of control.

Commas

Commas separate items that appear in lists. We can create multiple variables of the same type. E.g.,

```
unsigned short beginTime,endTime,elapsedTime;
```

Lists are also used with functions having multiple parameters (both when the function is defined and called):

```
short add(short x, short y){ short z;
  z = x+y;
  if((x>0)&&(y>0)&&(z<0))z = 32767;
  if((x<0)&&(y<0)&&(z>0))z = -32768;
  return(z);
}
void main(void){ short a,b;
  a=add(2000,2000)
  b=0
  while(1){
    b=add(b,1);
}
```

Listing 2-6: Commas separate the parameters of a function

Lists can also be used in general expressions. Sometimes it adds clarity to a program if related variables are modified at the same place. The value of a list of expressions is always the value of the last expression in the list. In the following example, first `thetime` is incremented, thedate is decremented, then x is set to k+2.

```
x=(thetime++,--thedate,k+2);
```

Apostrophes

Apostrophes are used to specify character literals. For more information about character literals see the section on characters in Chapter 3. Assuming the function `OutChar` will print a single ASCII character, the following example will print the lower case alphabet:

```
void Alphabet(void){ unsigned char mych;
  for(mych = 'a'; mych <= 'z'; mych++){
    OutChar(mych); /* Print next letter */
  }
}
```

Listing 2-7: Apostrophes are used to specify characters

Quotation marks

Quotation marks are used to specify string literals. For more information about string literals see the section on strings in Chapter 3. Example

```
unsigned const char Msg[12]= "Hello World";
/* Msg has 11 characters and termination*/
void PrintHelloWorld(void){
  UART_OutString("Hello World");
  UART_OutString(Msg);
}
```

Listing 2-8: Quotation marks are used to specify strings

The command `Letter='A';` places the ASCII code (65) into the variable `Letter`. The command `pt="A";` creates an ASCII string and places a pointer to it into the variable `pt`.

Braces

Braces {} are used throughout C programs. The most common application is for creating a compound statement. Each open brace { must be matched with a closing brace }. One approach that helps to match up braces is to use indenting. Each time an open brace is used, the source code is tabbed over. In this way, it is easy to see at a glance the brace pairs. Examples of this approach to tabbing are the Bit_Put function within Listing 2-2 and the median function in Listing 1-4.

Brackets

Square brackets enclose array *dimensions* (in declarations) and *subscripts* (in expressions). Thus,

```
short Fifo[100];
```

declares an integer array named `Fifo` consisting of 80 words numbered from 0 through 99, and

```
PutPt = &Fifo[0];
```

assigns the variable `PutPt` to the address of the first entry of the array.

Parentheses

Parentheses enclose argument lists that are associated with function declarations and calls. They are required even if there are no arguments.

As with all programming languages, C uses parentheses to control the order in which expressions are evaluated. Thus, (11+3)/2 yields 7, whereas 11+3/2 yields 12. Parentheses are very important when writing expressions.

Operators

The special characters used as *expression operators* are covered in the operator section in chapter 5. There are many operators, some of which are single characters

```
~  !  @  %  ^  &  *  -  +  =  |  /  :  ?  <  > ,
```

while others require two characters

```
++  --  <<  >>  <=  +=  -=  *=  /=  ==  |=  %=  &=  ^=  ||  &&  !=
```

and some even require three characters

```
<<=  >>=
```

The multiple-character operators can not have white spaces or comments between the characters.

The C syntax can be confusing to the beginning programmer. For example

```
z = x+y;   /* sets z equal to the sum of x and y */
z = x_y;   /* sets z equal to the value of x_y */
```

Go to Chapter 3 on Literals Return to Table of Contents

Chapter 3: Numbers, Characters and Strings

What's in Chapter 3?

This chapter defines the various data types supported by the compiler. Since the objective of most computer systems is to process data, it is important to understand how data is stored and interpreted by the software. We define a *literal* as the direct specification of the number, character, or string. E.g.,

```
        100 'a' "Hello World"
```

are examples of a number literal, a character literal and a string literal respectively. We will discuss the way data are stored on the computer as well as the C syntax for creating the literals. The Imagecraft and Metrowerks compilers recognize three types of literals (*numeric*, *character*, *string*). Numbers can be written in three bases (*decimal*, *octal*, and *hexadecimal*). Although the programmer can choose to specify numbers in these three bases, once loaded into the computer, the all numbers are stored and processed as unsigned or signed binary. Although C does not support the binary literals, if you wanted to specify a binary number, you should have no trouble using either the octal or hexadecimal format.

Binary representation

Numbers are stored on the computer in binary form. In other words, information is encoded as a sequence of 1's and 0's. On most computers, the memory is organized into 8-bit bytes. This means each 8-bit byte stored in memory will have a separate address. *Precision* is the number of distinct or different values. We express precision in alternatives, decimal digits, bytes, or binary bits. *Alternatives* are defined as the total number of possibilities. For example, an 8-bit number scheme can represent 256 different numbers. An 8-bit *digital to analog converter* can generate 256 different analog outputs. An 8-bit *analog to digital converter* (ADC) can measure 256 different analog inputs. We use the expression 4½ decimal digits to mean about 20,000 alternatives and the expression 4¾ decimal digits to mean more than 20,000 alternatives but less than 100,000 alternatives. The ½ decimal digit means twice the number of alternatives or one additional binary bit. For example, a voltmeter with a range of 0.00 to 9.99V has a three decimal digit precision. Let the operation [[x]] be the greatest integer of x. E.g., [[2.1]] is rounded up to 3. Tables 3.1a and 3.1b illustrate various representations of precision.

Binary bits	Bytes	Alternatives
8	1	256
10		1024
12		4096

16	2	65536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	[[n/8]]	2^n

Table 3-1a. Relationships between various representations of precision.

Decimal digits	**Alternatives**
3	1000
3½	2000
3¾	4000
4	10000
4½	20000
4¾	40000
5	100000
n	10^n

Table 3-1b. Relationships between various representations of precision.

Observation: A good rule of thumb to remember is $2^{10\bullet n}$ is about $10^{3\bullet n}$.

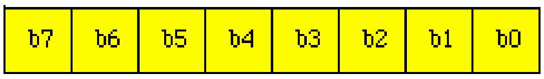
For large numbers we use abbreviations, as shown in the following table. For example, 16K means 16*1024 which equals 16384. Computer engineers use the same symbols as other scientists, but with slightly different values.

abbreviation	pronunciation	Computer Engineering Value	Scientific Value
K	"kay"	2^{10} 1024	10^3
M	"meg"	2^{20} 1,048,576	10^6
G	"gig"	2^{30} 1,073,741,824	10^9
T	"tera"	2^{40} 1,099,511,627,776	10^{12}
P	"peta"	2^{50} 1,125,899,906,843,624	10^{15}
E	"exa"	2^{60} 1,152,921,504,606,846,976	10^{18}

Table 3-2. Common abbreviations for large numbers.

8-bit unsigned numbers

A byte contains 8 bits



where each bit b7,...,b0 is binary and has the value 1 or 0. We specify b7 as the *most significant bit* or MSB, and b0 as the least significant bit or LSB. If a byte is used to represent an unsigned number, then the value of the number is

N = 128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

There are 256 different unsigned 8-bit numbers. The smallest unsigned 8-bit number is 0 and the largest is 255. For example, 00001010_2 is 8+2 or 10. Other examples are shown in the following table.

binary	hex	Calculation	decimal
00000000	0x00		0
01000001	0x41	64+1	65
00010110	0x16	16+4+2	22
10000111	0x87	128+4+2+1	135
11111111	0xFF	128+64+32+16+8+4+2+1	255

Table 3-3. Example conversions from unsigned 8-bit binary to hexadecimal and to decimal.

The *basis* of a number system is a subset from which linear combinations of the basis elements can be used to construct the entire set. For the unsigned 8-bit number system, the basis is

{ 1, 2, 4, 8, 16, 32, 64, 128}

One way for us to convert a decimal number into binary is to use the basis elements. The overall approach is to start with the largest basis element and work towards the smallest. One by one we ask ourselves whether or not we need that basis element to create our number. If we do, then we set the corresponding bit in our binary result and subtract the basis element from our number. If we do not need it, then we clear the corresponding bit in our binary result. We will work through the algorithm with the example of converting 100 to 8 bit binary. We with the largest basis element (in this case 128) and ask whether or not we need to include it to make 100. Since our number is less than 128, we do not need it so bit 7 is zero. We go the next largest basis element, 64 and ask do we need it. We do need 64 to generate our 100, so bit 6 is one and subtract 100 minus 64 to get 36. Next we go the next basis element, 32 and ask do we need it. Again we do need 32 to generate our 36, so bit 5 is one and we subtract 36 minus 32 to get 4. Continuing along, we need basis element 4 but not 16 8 2 or 1, so bits 43210 are 00100 respectively. Putting it together we get 011001002 (which means 64+32+4).

Observation: If the least significant binary bit is zero, then the number is even.

Observation: If the right most n bits (least significant) are zero, then the number is divisible by 2^n.

Number	Basis	Need it	bit	Operation
100	128	no	bit7=0	none
100	64	yes	bit6=1	subtract 100-64
36	32	yes	bit5=1	subtract 36-32

4	16	no	bit4=0	none
4	8	no	bit3=0	none
4	4	yes	bit2=1	subtract 4-4
0	2	no	bit1=0	none
0	1	no	bit0=0	none

Table 3-4. Example conversion from decimal to unsigned 8-bit binary to hexadecimal.

We define an unsigned 8-bit number using the `unsigned char` format. When a number is stored into an `unsigned char` it is converted to 8-bit unsigned value. For example

```
unsigned char data; // 0 to 255
unsigned char function(unsigned char input){
    data=input+1;
    return data;}
```

8-bit signed numbers

If a byte is used to represent a *signed 2's complement* number, then the value of the number is

N = -128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

There are also 256 different signed 8 bit numbers. The smallest signed 8-bit number is -128 and the largest is 127. For example, 10000010_2 is -128+2 or -126. Other examples are shown in the following table.

binary	hex	Calculation	decimal
00000000	0x00		0
01000001	0x41	64+1	65
00010110	0x16	16+4+2	22
10000111	0x87	-128+4+2+1	-121
11111111	0xFF	-128+64+32+16+8+4+2+1	-1

Table 3-5. Example conversions from signed 8-bit binary to hexadecimal and to decimal.

For the signed 8-bit number system the basis is

{ 1, 2, 4, 8, 16, 32, 64, -128}

Observation: The most significant bit in a 2's complement signed number will specify the sign.

Notice that the same binary pattern of 11111111_2 could represent either 255 or -1. It is very important for the software developer to keep track of the number format. The computer can not determine whether the 8-bit number is signed or unsigned. You, as the programmer, will determine whether the number is signed or unsigned by the specific assembly instructions you select to operate on the number. Some operations like addition, subtraction, and shift left (multiply by 2) use the same hardware (instructions) for both unsigned and signed operations. On the other hand, multiply, divide, and shift right (divide by 2) require separate hardware (instruction) for unsigned and signed operations. The compiler will automatically choose the proper implementation.

It is always good programming practice to have clear understanding of the data type for each number, variable, parameter, etc. For some operations there is a difference between the signed and unsigned numbers while for others it does not matter.

	signed different from unsigned		signed same as unsigned
/ %	division	+	addition
*	multiplication	-	subtraction
>	greater than	==	is equal to
<	less than	|	logical or
>=	greater than or equal to	&	logical and
<=	less than or equal to	^	logical exclusive or
>>	right shift	<<	left shift

Table 3-6. Operations either depend or don't depend on whether the number is signed/unsigned.

The point is that care must be taken when dealing with a mixture of numbers of different sizes and types.

Similar to the unsigned algorithm, we can use the basis to convert a decimal number into signed binary. We will work through the algorithm with the example of converting -100 to 8-bit binary. We with the largest basis element (in this case -128) and decide do we need to include it to make -100. Yes (without -128, we would be unable to add the other basis elements together to get any negative result), so we set bit 7 and subtract the basis element from our value. Our new value is -100 minus -128, which is 28. We go the next largest basis element, 64 and ask do we need it. We do not need 64 to generate our 28, so bit6 is zero. Next we go the next basis element, 32 and ask do we need it. We do not need 32 to generate our 28, so bit5 is zero. Now we need the basis element 16, so we set bit4, and subtract 16 from our number 28 (28-16=12). Continuing along, we need basis elements 8 and 4 but not 2 1, so bits 3210 are 1100. Putting it together we get 100111002 (which means -128+16+8+4).

Number	Basis	Need it	bit	Operation
-100	-128	yes	bit7=1	subtract -100 - -128
28	64	no	bit6=0	none
28	32	no	bit5=0	none
28	16	yes	bit4=1	subtract 28-16
12	8	yes	bit3=1	subtract 12-8
4	4	yes	bit2=1	subtract 4-4
0	2	no	bit1=0	none
0	1	no	bit0=0	none

Table 3-7. Example conversion from decimal to signed 8-bit binary to hexadecimal.

Observation: To take the negative of a 2's complement signed number we first complement (flip) all the bits, then add 1.

A second way to convert negative numbers into binary is to first convert them into unsigned binary, then do a 2's complement negate. For example, we earlier found that +100 is 011001002. The 2's complement negate is a two step process. First we do a logic complement (flip all bits) to get 100110112. Then add one to the result to get 100111002.

A third way to convert negative numbers into binary is to first subtract the number from 256, then convert the unsigned result to binary using the unsigned method. For example, to find -100, we subtract 256 minus 100 to get 156. Then we convert 156 to binary resulting in 100111002. This method works because in 8 bit binary math adding 256 to number does not change the value. E.g., 256-100 is the same value as -100.

> *Common Error: An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers.*

> *Maintenance Tip: To improve the clarity of our software, always specify the format of your data (signed versus unsigned) when defining or accessing the data.*

We define a signed 8-bit number using the `char` format. When a number is stored into a `char` it is converted to 8-bit signed value. For example

```
char data; // -128 to 127
char function(char input){
    data=input+1;
    return data;}
```

16 bit unsigned numbers

A *halfword* or *double byte* contains 16 bits. A *word* contains 32 bits.

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

where each bit b15,...,b0 is binary and has the value 1 or 0. If a word is used to represent an unsigned number, then the value of the number is

N = 32768•b15 + 16384•b14 + 8192•b13 + 4096•b12

+ 2048•b11 + 1024•b10 + 512•b9 + 256•b8

+ 128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

There are 65,536 different unsigned 16-bit numbers. The smallest unsigned 16-bit number is 0 and the largest is 65535. For example, $0010,0001,1000,0100_2$ or 0x2184 is 8192+256+128+4 or 8580. Other examples are shown in the following table.

binary	hex	Calculation	decimal
0000,0000,0000,0000	0x0000		0
0000,0100,0000,0001	0x0401	1024+1	1025
0000,1100,1010,0000	0x0CA0	2048+1024+128+32	3232
1000,1110,0000,0010	0x8E02	32768+2048+1024+512+2	36354

1111,1111,1111,1111	0xFFFF	32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+4+2+1	65535

Table 3-8. Example conversions from unsigned 16-bit binary to hexadecimal and to decimal.

For the unsigned 16-bit number system the basis is

{ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768}

If a word is used to represent a signed 2's complement number, then the value of the number is

N = -32768•b15 + 16384•b14 + 8192•b13 + 4096•b12

+ 2048•b11 + 1024•b10 + 512•b9 + 256•b8

+ 128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

We define an unsigned 16-bit number using the `unsigned short` format. When a number is stored into an `unsigned short` it is converted to 16-bit unsigned value. For example

```
unsigned short data; // 0 to 65535
unsigned short function(unsigned short input){
    data=input+1;
    return data;}
```

16-bit signed numbers

There are also 65,536 different signed 16-bit numbers. The smallest signed 16-bit number is -32768 and the largest is 32767. For example, $1101,0000,0000,0100_2$ or 0xD004 is -32768+16384+4096+4 or -12284. Other examples are shown in the following table.

binary	hex	Calculation	decimal
0000,0000,0000,0000	0x0000		0
0000,0100,0000,0001	0x0401	1024+1	1025
0000,1100,1010,0000	0x0CA0	2048+1024+128+32	3232
1000,0100,0000,0010	0x8402	-32768+1024+2	-31742
1111,1111,1111,1111	0xFFFF	-32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+4+2+1	-1

Table 3-9. Example conversions from signed 16-bit binary to hexadecimal and to decimal.

For the signed 16-bit number system the basis is

{ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, -32768}

Maintenance Tip: To improve the quality of our software, we should always specify the precision of our data when defining or accessing the data.

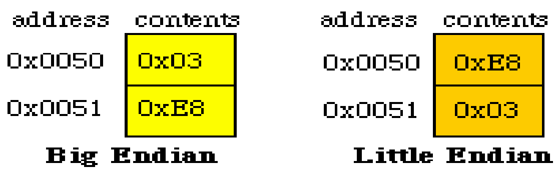
We define a signed 16-bit number using the `short` format. When a number is stored into a `short` it is converted to 16-bit signed value. For example

```
short data; // -23768 to 32767
```

```
        short function(short input){
            data=input+1;
            return data;}
```

Big and Little Endian

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the *big endian* approach that stores the most significant part first. The ARM Cortex M processors implement the *little endian* approach that stores the least significant part first. Some ARM processors are *biendian*, because they can be configured to efficiently handle both big and little endian. For example, assume we wish to store the 16 bit number 1000 (0x03E8) at locations 0x50,0x51, then



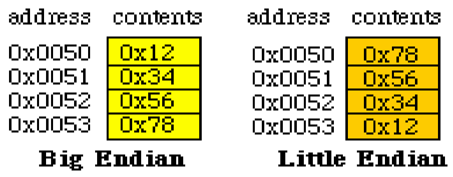
address	contents
0x0050	0x03
0x0051	0xE8

Big Endian

address	contents
0x0050	0xE8
0x0051	0x03

Little Endian

We also can use either the big or little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. If we wish to store the 32-bit number 0x12345678 at locations 0x50-0x53 then



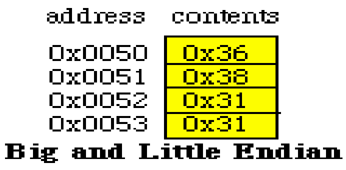
address	contents
0x0050	0x12
0x0051	0x34
0x0052	0x56
0x0053	0x78

Big Endian

address	contents
0x0050	0x78
0x0051	0x56
0x0052	0x34
0x0053	0x12

Little Endian

In the above two examples we normally would not pick out individual bytes (e.g., the 0x12), but rather capture the entire multiple byte data as one nondivisable piece of information. On the other hand, if each byte in a multiple byte data structure is individually addressable, then both the big and little endian schemes store the data in first to last sequence. For example, if we wish to store the 4 ASCII characters ‘6811’ which is 0x36383131 at locations 0x50-0x53, then the ASCII ‘6’=0x36 comes first in both big and little endian schemes.



address	contents
0x0050	0x36
0x0051	0x38
0x0052	0x31
0x0053	0x31

Big and Little Endian

The term "Big Endian" comes from Jonathan Swift’s satire Gulliver’s Travels. In Swift’s book, a

Big Endian refers to a person who cracks their egg on the big end. The Lilliputians considered the big endians as inferiors. The big endians fought a long and senseless war with the Lilliputians who insisted it was only proper to break an egg on the little end.

Common Error: An error will occur when data is stored in Big Endian by one computer and read in Little Endian format on another.

Boolean information

A boolean number is has two states. The two values could represent the logical true or false. The positive logic representation defines true as a 1 or high, and false as a 0 or low. If you were controlling a motor, light, heater or air conditioner the boolean could mean on or off. In communication systems, we represent the information as a sequence of booleans: mark or space. For black or white graphic displays we use booleans to specify the state of each pixel. The most efficient storage of booleans on a computer is to map each boolean into one memory bit. In this way, we could pack 8 booleans into each byte. If we have just one boolean to store in memory, out of convenience we allocate an entire byte or word for it. Most C compilers including Keil uVision define:

False be all zeros, and
True be any nonzero value.

Many programmers add the following macros

```
#define TRUE 1
#define FALSE 0
```

Decimal Numbers

Decimal numbers are written as a sequence of decimal digits (0 through 9). The number may be preceded by a plus or minus sign or followed by a **L**or **U**. Lower case **l** or **u** could also be used. The minus sign gives the number a negative value, otherwise it is positive. The plus sign is optional for positive values. Unsigned 16-bit numbers between 32768 and 65535 should be followed by **U**. You can place a **L** at the end of the number to signify it to be a 32-bit signed number. The range of a decimal number depends on the data type as shown in the following table. On the Keil uVision compiler, the **char** data type may be signed or unsigned depending on a compiler option.

type	range	precision	examples
unsigned char	0 to 255	8 bits	0 10 123
char	-128 to 127	8 bits	-123 0 10 +10
unsigned int	0 to 4294967295	32 bits	0 2000 2000 50000000L
int	-2147483648 to 2147483647	16 bits	-1000 0 1000 +20000
unsigned short	0 to 65535U	16 bits	0 2000 2000U 50000U
short	-32768 to 32767	16 bits	-1000 0 1000 +20000
long	-2147483648 to 2147483647	32 bits	-123456L 0L 1234567L
unsigned long	0 to 4294967295	32 bits	0L 12345678L

Table 3-10. The range of decimal numbers.

Because the ARM Cortex microcomputers are most efficient for 32 bit-data (and not 64-bit data), the **unsigned int** and **int** data types are 32 bits. On the other hand, on a 9S16-based machine, the **unsigned int** and **int** data types are 16 bits. In order to make your software more compatible with other machines, it is preferable to use the **short** type when needing 16-bit data and the **long** type for 32-bit data.

type	6811/6812	Cortex M
unsigned char	8 bits	8 bits
char	8 bits	8 bits
unsigned int	16 bits	32 bits
int	16 bits	32 bits
unsigned short	16 bits	16 bits
short	16 bits	16 bits
long	32 bits	32 bits

Table 3-11. Differences between a 6811/6812 and an ARM Cortex M

Since the Cortex M microcomputers do not have direct support of 64-bit numbers, the use of **long long** data types should be minimized. On the other hand, a careful observation of the code generated yields the fact that these compilers are more efficient with 32-bit numbers than with 8-bit or 16-bit numbers.

Decimal numbers are reduced to their two's complement or unsigned binary equivalent and stored as 8/16/32-bit binary values.

The manner in which decimal literals are treated depends on the context. For example

```
short I;
unsigned short J;
char K;
unsigned char L;
long M;
void main(void){
    I=97;    /* 16 bits 0x0061 */
    J=97;    /* 16 bits 0x0061 */
    K=97;    /* 8 bits 0x61 */
    L=97;    /* 8 bits 0x61 */
    M=97;    /* 32 bits 0x00000061 */}
```

Octal Numbers

If a sequence of digits begins with a leading **0**(zero) it is interpreted as an octal value. There are only eight octal digits, 0 through 7. As with decimal numbers, octal numbers are converted to their binary equivalent in 8-bit or 16-bit words. The range of an octal number depends on the data type as shown in the following table.

type	range	precision	examples
unsigned char	0 to 0377	8 bits	0 010 0123
char	-0200 to 0177	8 bits	-0123 0 010 +010
unsigned short	0 to 0177777	16 bits	0 02000 0150000U
short	-0100000 to 077777	16 bits	-01000 0 01000 +020000
long	-020000000000 to 017777777777	32 bits	-01234567L 0L 01234567L

Table 3-12. The range of octal numbers.

Notice that the octal values 0 through 07 are equivalent to the decimal values 0 through 7. One of the advantages of this format is that it is very easy to convert back and forth between octal and binary. Each octal digit maps directly to/from 3 binary digits.

Hexadecimal Numbers

The hexadecimal number system uses base 16 as opposed to our regular decimal number system that uses base 10. Like the octal format, the hexadecimal format is also a convenient mechanism for us humans to represent binary information, because it is extremely simple for us to convert back and forth between binary and hexadecimal. A *nibble* is defined as 4 binary bits. Each value of the 4-bit nibble is mapped into a unique hex digit.

Hex Digit	Decimal Value	Binary Value
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A or a	10	1010
B or b	11	1011
C or c	12	1100
D or d	13	1101
E or e	14	1110
F or f	15	1111

Table 3-13. Definition of hexadecimal representation.

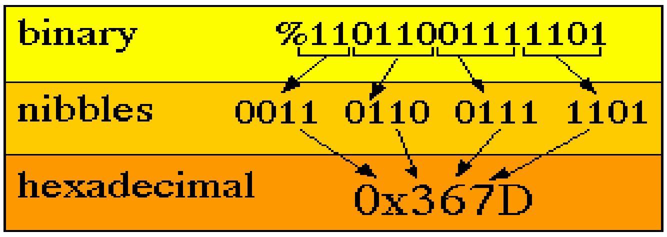
Computer programming environments use a wide variety of symbolic notations to specify the numbers in various bases. The following table illustrates various formats for numbers

environment	binary format	hexadecimal format	decimal format
Freescale assembly language	%01111010	$7A	122
Intel and TI assembly language	01111010B	7AH	122
C language	-	0x7A	122

Table 3-14. Various hexadecimal formats.

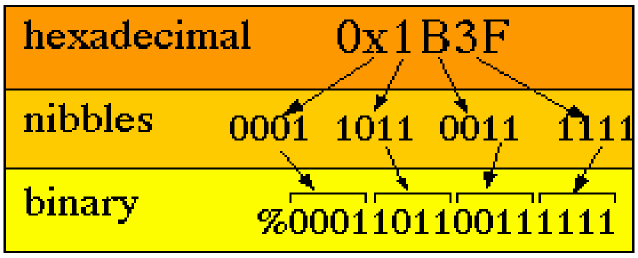
To convert from binary to hexadecimal we can:

1) divide the binary number into right justified nibbles;
2) convert each nibble into its corresponding hexadecimal digit.



To convert from hexadecimal to binary we can:

1) convert each hexadecimal digit into its corresponding 4 bit binary nibble;
2) combine the nibbles into a single binary number.



If a sequence of digits begins with **0x** or **0X** then it is taken as a hexadecimal value. In this case the word digits refers to hexadecimal digits (0 through F). As with decimal numbers, hexadecimal numbers are converted to their binary equivalent in 8-bit bytes or16-bit words. The range of a hexadecimal number depends on the data type as shown in the following table.

type	range	precision	examples
unsigned char	0x00 to 0xFF	8 bits	0x01 0x3a 0xB3
char	-0x80 to 0x7F	8 bits	-0x01 0x3a -0x7B
unsigned short	0x0000 to 0xFFFF	16 bits	0x22 0Xabcd 0xF0A6
short	-0x8000 to 0x7FFF	16 bits	-0x1234 0x0 +0x7abc
long	-0x80000000 to 0x7FFFFFFF	32 bits	-0x1234567 0xABCDEF

Table 3-15. The range of hexadecimal numbers.

Character Literals

Character literals consist of one or two characters surrounded by apostrophes. The manner in which character literals are treated depends on the context. For example

```
short I;
unsigned short J;
char K;
unsigned char L;
long M;
void main(void){
    I='a';    /* 16 bits 0x0061 */
    J='a';    /* 16 bits 0x0061 */
    K='a';    /* 8 bits 0x61 */
    L='a';    /* 8 bits 0x61 */
    M='a';    /* 32 bits 0x00000061 */}
```

All standard ASCII characters are positive because the high-order bit is zero. In most cases it doesn't matter if we declare character variables as signed or unsigned. On the other hand, we have seen earlier that the compiler treats signed and unsigned numbers differently. Unless a character variable is specifically declared to be unsigned, its high-order bit will be taken as a sign bit. Therefore, we should not expect a character variable, which is not declared unsigned, to compare equal to the same character literal if the high-order bit is set. For more on this see Chapter 4 on Variables.

String Literals

Strictly speaking, C does not recognize character strings, but it does recognize arrays of characters and provides a way to write character arrays, which we call *strings*. Surrounding a character sequence with quotation marks, e.g., **"Jon"**, sets up an array of characters and generates the address of the array. In other words, at the point in a program where it appears, a string literal produces the address of the specified array of character literals. The array itself is located elsewhere. Metrowerks will place strings into the text area. I.e., the string literals are considered constant and will be defined in the ROM of an embedded system. This is very important to remember. Notice that this differs from a character literal which generates the value of the literal directly. Just to be sure that this distinct feature of the C language is not overlooked, consider the following example:

```
char *pt;
extern void Foo(char *p);
void main(void){
    pt="Jon"; /* pointer to the string */
    Foo(pt); /* passes the pointer not the data itself */
}
```

Note that the pointer, `pt`, is allocated in RAM and the string is stored in ROM. The assignment statement `pt="Jon";` copies the address not the data. First, the address of the string is assigned to the character pointer `pt` (Keil uVision uses the 32-bit Register R0 for the first parameter). Unlike other languages, the string itself is not assigned to `pt`, only its address is. After all, `pt` is a 32-bit object and, therefore, cannot hold the string itself.

Since strings may contain as few as one or two characters, they provide an alternative way of writing character literals in situations where the address, rather than the character itself, is needed.

It is a convention in C to identify the end of a character string with a null (zero) character. Therefore, C compilers automatically suffix character strings with such a terminator. Thus, the string **"Jon"** sets up an array of four characters (**'J'**, **'o'**, **'n'**, and zero) and generates the address of the first character, for use by the program.

Remember that 'A' is different from "A", consider the following example:

```
char letter,*pt;
void main(void){
    pt="A";      /* pointer to the string */
    letter='A';  /* the data itself ('A' ASCII 65=$41) */
}
```

Escape Sequences

Sometimes it is desirable to code nongraphic characters in a character or string literal. This can be done by using an *escape sequence*--a sequence of two or more characters in which the first (escape) character changes the meaning of the following character(s). When this is done the entire sequence generates only one character. C uses the backslash (\) for the escape character. The following escape sequences are recognized by the Metrowerks compiler:

sequence	name	value
\n	newline, linefeed	0x0A = 10
\t	tab	0x09 = 9
\b	backspace	0x08 = 8

\f	form feed	0x0C = 12
\a	bell	0x07 = 7
\r	return	0x0D = 13
\v	vertical tab	0x0B = 11
\0	null	0x00 = 0
\"	ASCII quote	0x22 = 34
\\	ASCII back slash	0x5C = 92
\'	ASCII single quote	0x27 = 39

Table 3-16. The escape sequences supported by Keil uVision.

Other nonprinting characters can also be defined using the \ooo octal format. The digits *ooo* can define any 8-bit octal number. The following three lines are equivalent:

```
printf("\tJon\n");
printf("\11Jon\12");
printf("\011Jon\012");
```

The term *newline* refers to a single character which, when written to an output device, starts a new line. Some hardware devices use the ASCII carriage return (13) as the newline character while others use the ASCII line feed (10). It really doesn't matter which is the case as long as we write `\n` in our programs. Avoid using the ASCII value directly since that could produce compatibility problems between different compilers.

There is one other type of escape sequence: anything undefined. If the backslash is followed by any character other than the ones described above, then the backslash is ignored and the following character is taken literally. So the way to code the backslash is by writing a pair of backslashes and the way to code an apostrophe or a quote is by writing \' or \" respectively.

Go to Chapter 4 on Variables Return to Table of Contents

Chapter 4: Variables and Constants

What's in Chapter 4?

The purpose of this chapter is to explain how to create and access variables and constants. The storage and retrieval of information are critical operations of any computer system. This chapter will also present the C syntax and resulting assembly code generated by the Keil uVision compiler.

A *variable* is a named object that resides in RAM memory and is capable of being examined and modified. A variable is used to hold information critical to the operation of the embedded system. A *constant* is a named object that resides in memory (usually in ROM) and is only capable of being examined. As we saw in the last chapter a *literal* is the direct specification of a number character or string. The difference between a literal and a constant is that constants are given names so that they can be accessed more than once. For example

```
short MyVariable;          /* variable allows read/write access */
const short MyConstant=50; /* constant allows only read access */
#define fifty 50
void main(void){
    MyVariable=50;         /* write access to the variable */
    OutSDec(MyVariable);   /* read access to the variable */
    OutSDec(MyConstant);   /* read access to the constant */
    OutSDec(50);           /* "50" is a literal */
    OutSDec(fifty);        /* fifty is also a literal */
}
```

Listing 4-1: Example showing a variable, a constant and some literals

The options on many compilers can be used to select the precision of each of the data formats (**int**, **short** etc.)

The concepts of precision and type (unsigned vs. signed) developed for numbers in the last chapter apply to variables and constants as well. In this chapter we will begin the discussion of variables that contain integers and characters. Even though pointers are similar in many ways to 32-bit unsigned integers, pointers will be treated in detail in Chapter 7. Although arrays and structures fit also the definition of a variable, they are regarded as collections of variables and will be discussed in Chapter 8 and Chapter 9.

The term *storage class* refers to the method by which an object is assigned space in memory. The Metrowerks compiler recognizes three storage classes--static, automatic, and external. In this document we will use the term *global variable* to mean a regular static variable that can be accessed by all other functions. Similarly we will use the term *local variable* to mean an automatic variable that can be accessed only by the function that created it. As we will see in the following sections there are other possibilities like a static global and static local.

Statics

Static variables are given space in memory at some fixed location within the program. They exist when the program starts to execute and continue to exist throughout the program's entire lifetime. The value of a static variable is faithfully maintained until we change it deliberately (or remove power from the memory). A constant, which we define by adding the modifier **const**, can be read but not changed.

In an embedded system we normally wish to place all variables in RAM and constants in ROM. The following example sets a global, called TheGlobal, to the value 1000. This global can be referenced by any function from any file in the software system. It is truly global.

```
long TheGlobal;   /* a regular global variable*/
void main(void){
  TheGlobal = 1000;
}
```

Listing 4-2: Example showing a regular global variable

The ARM code generated by the uVision compiler is as follows (register use will vary)

```
    LDR R0,=1000
    LDR R1,=TheGlobal
    STR R0,[R1]
```

The fact that these types of variables exist in permanently reserved memory means that static variables exist for the entire life of the program. When the power is first applied to an embedded computer, the values in its RAM are usually undefined. Therefore, initializing global variables requires special run-time software consideration. See the section on initialization for more about initializing variables and constants.

A **static global** is very similar to a regular global. In both cases, the variable is defined in RAM permanently. The assembly language access is identical. The only difference is the scope. The static global can only be accessed within the file where it is defined. The following example also sets a global, called **TheGlobal**, to the value 1000. This global can not be referenced by modules in other files.

```
static short TheGlobal;   /* a static global variable*/
void main(void){
  TheGlobal = 1000;
}
```

Listing 4-3: Example showing a static global variable

The code generated by the compiler is the *same as* a regular global. The compiler does properly limit the access only to the static global to functions defined in this file.

A **static local** is similar to the static global. Just as with the other statics, the variable is defined in RAM permanently. The assembly language code generated by the compiler that accesses the variable is identical. The only difference is the scope. The static local can only be accessed within the function where it is defined. The following example sets a static local, called **TheLocal**, to the value 1000. The compiler limits the access to the static local, so that this variable can not be accessed by other functions in this file or in other files. Notice that the assembly language name of the static local is a unique compiler-generated name (L2 in this example.) This naming method allows other functions to also define a static local or automatic local with the same name.

```
void main(void){
  static stort TheLocal;   /* a static local variable*/
  TheLocal = 1000;
}
```

Listing 4-4: Example showing a static local variable

Again the code generated by the compiler is the same as a regular global. The compiler does properly limit the access only to the static local to the function in which it is defined.

A **static local** can be used to save information from one instance of the function call to the next. Assume each function wished to know how many times it has been called. Remember upon reset, the compiler will initialize all statics to zero (including static locals). The following functions maintain such a count, and these counts can not be accessed by other functions. Even though the names are the same, the two static locals are in fact distinct.

```
void function1(void){
  static short TheCount;
  TheCount = TheCount+1;
}
void function2(void){
  static short TheCount;
  TheCount = TheCount+1;
}
```

Listing 4-5: Example showing two static local variables with the same name

Volatile

We add the **volatile** modifier to a variable that can change value outside the scope of the function. Usually the value of a global variable changes only as a result of explicit statements in the C function that is currently executing. The paradigm results when a single program executes from start to finish, and everything that happens is an explicit result of actions taken by the program. There are two situations that break this simple paradigm in which the value of a memory location might change outside the scope of a particular function currently executing:

1) interrupts and
2) input/output ports.

An interrupt is a hardware-requested software action. Consider the following multithreaded interrupt example. There is a foreground thread called **main()**, which we setup as the usual main program that all C programs have. Then, there is a background thread called **SysTick_Handler()**, which we setup to be executed on a periodic basis (e.g., every 16 ms). Both threads access the global variable, **Time**. The interrupt thread increments the global variable, and the foreground thread waits for time to reach 100. Notice that **Time** changes value outside the influence of the **main()** program.

```
volatile unsigned long Time;
void SysTick_Handler(void){    /* every 16ms */
```

```
    Time = Time+1;
  }
  void main(void){
    SysTick_Init();
    Time = 0;
    while(Time<100){}; /* wait for 100 counts of the 16 ms timer*/
  }
```

Listing 4-6: An example showing shared access to a common global variable

Without the **volatile** modifier the compiler might look at the two statements:

```
  Time = 0;
  while(Time<100){};
```

and conclude that since the while loop does not modify **Time**, it could never reach 100. Some compilers might attempt to move the read **Time** operation, performing it once before the while loop is executed. The **volatile** modifier disables the optimization, forcing the program to fetch a new value from the variable each time the variable is accessed.

In the next example, assume PORTA is an input port containing the current status of some important external signals. The program wishes to collect status versus time data of these external signals.

```
  unsigned char data[100];
  #define GPIO_PORTA_DATA_R       (*((volatile unsigned long *)0x400043FC))
  void Collect(void){ short i;
    for(i=0;i<100;i++){ /* collect 100 measurements */
      data[i] = GPIO_PORTA_DATA_R;  /* collect ith measurement */
    }
  }
```

Listing 4-7: Example showed shared access to a common global variable

Without the **volatile** modifier in the PORTA definition, the compiler might optimize the for loop, reading PORTA once, then storing 100 identical copies into the data array. I/O ports will be handled in more detail in Chapter 7 on pointers.

Automatics

Automatic variables, on the other hand, do not have fixed memory locations. They are dynamically allocated when the block in which they are defined is entered, and they are discarded upon leaving that block. When there are a few variables, they are allocated in registers. However, when there are a lot of local variables, they are allocated on the stack by subtracting 4 from the SP for each 32-bit variable. Since automatic objects exist only within blocks, they can only be declared locally. Automatic variables can only be referenced (read or write) by the function that created it. In this way, the information is protected or local to the function.

When a local variable is created it has no dependable initial value. It must be set to an initial value by means of an assignment operation. C provides for automatic variables to be initialized in their declarations, like globals. It does this by generating "hidden" code that assigns values automatically after variables are allocated space.

It is tempting to forget that automatic variables go away when the block in which they are defined exits. This sometimes leads new C programmers to fall into the "dangling reference" trap in which a function returns a pointer to a local variable, as illustrated by

```
  int *BadFunction(void) {
   int z;
   z = 1000;
   return (&z);
}
```

Listing 4-8: Example showing an illegal reference to a local variable

When callers use the returned address of **z** they will find themselves messing around with the stack space that **z** used to occupy. This type of error is NOT flagged as a syntax error, but rather will cause unexpected behavior during execution.

Implementation of automatic variables

For information on how local variables are implemented on the ARM Cortex M see Chapter 7 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano. If locals are dynamically allocated at unspecified memory (stack) locations, then how does the program find them? This is done by using the stack pointer (SP) to designate a stack frame for the currently active function.

```
void fun(void){ long y1,y2,y3;   /* 3 local variables*/
    y1 = 1000;
    y2 = 2000;
    y3 = y1+y2;
}
fun SUB SP,#12 ; allocate3 local variables
;  y1 = 1000
    LDR R0,=1000
    STR R0,[SP,#0]
;y2 = 2000
    LDR R0,=2000
    STR R0,[SP,#4]
; y3 = y1+y2
    LDR R0,[SP,#0] ; y1
    LDR R1,[SP,#4] ; y2
    ADD R2,R0,R1
    STR R2,[SP,#8] ;set y3
    ADD SP,#12   ;deallocate
    BX  LR
```

Listing 4-9: Example showing three local variables

A **constant local** is similar to the regular local. Just as with the other locals, the constant is defined temporarily on the stack. The difference is that the constant local can not be changed. The assembly language code generated by the compiler that accesses the constant local is identical to the regular local.

```
short TheGlobal;   /* a regular global variable*/
void main(void){
    const short TheConstant=1000;   /* a constant local*/
    TheGlobal=TheConstant;
}
```

Listing 4-10: Example showing a constant local

Externals

Objects that are defined outside of the present source module have the external storage class. This means that, although the compiler knows what they are (signed/unsigned, 8-bit 16-bit 32-bit etc.), it has no idea where they are. It simply refers to them by name without reserving space for them. Then when the linker brings together the object modules, it resolves these "pending" references by finding the external objects and inserting their addresses into the instructions that refer to them. The compiler knows an external variable by the keyword **extern** that must precede its declaration.

Only global declarations can be designated extern and only globals in other modules can be referenced as external.

The following example sets an external global, called **ExtGlobal**, to the value 1000. This global can be referenced by any function from any file in the software system. It is truly global.

```
extern short ExtGlobal;   /* an external global variable*/
void main(void){
    ExtGlobal=1000;
}
```

Listing 4-11: Example showing an external global

Scope

The *scope* of a variable is the portion of the program from which it can be referenced. We might say that a variable's scope is the part of the program that "knows" or "sees" the variable. As we shall see, different rules determine the scopes of global and local objects.

When a variable is declared globally (outside of a function) its scope is the part of the source file that follows the declaration--any function following the declaration can refer to it. Functions that precede the declaration cannot refer to it. Most C compilers would issue an error message in that case.

The scope of local variables is the block in which they are declared. Local declarations must be grouped together before the first executable statement in the block--at the head of the block. This is different from C++ that allows local variables to be declared anywhere in the function. It follows that the scope of a local variable effectively includes all of the block in which it is declared. Since blocks can be nested, it also follows that local variables are seen in all blocks that are contained in the one that declares the variables.

If we declare a local variable with the same name as a global object or another local in a superior block, the new variable temporarily supersedes the higher level declarations. Consider the following program.

```
unsigned char x;   /* a regular global variable*/
void sub(void){
    x=1;
    {   unsigned char x;   /* a local variable*/
        x=2;
        {   unsigned char x;  /* a local variable*/
            x=3;
            PORTA=x;}
        PORTA=x;}
    PORTA=x;}
}
```

Listing 4-12: An example showing the scope of local variables

This program declares variables with the name **x**, assigns values to them, and outputs them to PORTA with in such a way that, when we consider its output, the scope of its declarations becomes clear. When this program runs, it outputs 321. This only makes sense if the **x** declared in the inner most block masks the higher level declarations so that it receives the value '3' without destroying the higher level variables. Likewise the second **x** is assigned '2' which it retains throughout the execution of the inner most block. Finally, the global **x**, which is assigned '1', is not affected by the execution of the two inner blocks. Notice, too, that the placement of the last two **PORTA=x;** statements demonstrates that leaving a block effectively unmasks objects that were hidden by declarations in the block. The second **PORTA=x;** sees the middle **x** and the last **PORTA=x;** sees the global **x**.

This masking of higher level declarations is an advantage, since it allows the programmer to declare local variables for temporary use without regard for other uses of the same names.

One of the mistakes a C++ programmer makes when writing C code is trying to define local variables in the middle of a block. In C local variables must be defined at the beginning of a block. The following example is proper C++ code, but results in a syntax error in C.

```
void sub(void){ int x;  /* a valid local variable declaration */
    x=1;
    int y;   /* This declaration is improper */
    y=2;
}
```

Listing 4-13: Example showing an illegal local variable declaration

Declarations

Unlike BASIC and FORTRAN, which will automatically declare variables when they are first used, every variable in C must be declared first. This may seem unnecessary, but when we consider how much time is spent debugging BASIC and FORTRAN programs simply because misspelled variable names are not caught for us, it becomes obvious that the time spent declaring variables beforehand is time well spent. Declarations also force us to consider the precision (8-bit, 16-bit etc.) and format (unsigned vs. signed) of each variable.

As we saw in Chapter 1, describing a variable involves two actions. The first action is declaring its type and the second action is defining it in memory (reserving a place for it). Although both of these may be involved, we refer to the C construct that accomplishes them as a *declaration*. As we saw above, if the declaration is preceded by **extern** it only declares the type of the variables, without reserving space for them. In such cases, the definition must exist in another source file. Failure to do so, will result in an unresolved reference error at link time.

Table 4-1 contains examples of legitimate variable declarations. Notice that the declarations are introduced by one or type keywords that states the data type of the variables listed. The keyword **char** declares 8-bit values, **int** declares 16-bit values, **short** declares 16-bit values and **long** declares 32-bit values. Unless the modifier **unsigned** is present, the variables declared by these statements are assumed by the compiler to contain signed values. You could add the keyword **signed** before the data type to clarify its type.

When more than one variable is being declared, they are written as a list with the individual names separated by commas. Each declaration is terminated with a semicolon as are all simple C statements.

Declaration	Comment	Range

unsigned char uc;	8-bit unsigned number	0 to +255
char c1,c2,c3;	three 8-bit signed numbers	-128 to +127
unsigned int ui;	32-bit unsigned number	0 to +4294967296
int i1,i2;	two 32-bit signed numbers	-2147483648L to 2147483647L
unsigned short us;	16-bit unsigned number	0 to +65535
short s1,s2;	two 16-bit signed numbers	-32768 to +32767
long l1,l2,l3,l4;	four signed 32 bit integers	-2147483648L to 2147483647L
unsigned long ui;	32-bit unsigned number	0 to +4294967296
float f1,f2;	two 32-bit floating numbers	not recommended
double d1,d2;	two 64-bit floating numbers	not recommended

Table 4-1: Variable Declarations

The compiler allows the **register** modifier for automatic variables, but it may still defines register locals on the stack. The keywords **char int short long** specify the precision of the variable. The following tables shows the available modifiers for variables.

Modifier	Comment
`auto`	automatic, allocated on the stack
`extern`	defined in some other program file
`static`	permanently allocated
`register`	attempt to implement an automatic using a register instead of on the stack

Table 4-2: Variable storage classes

Modifier	Comment
`volatile`	can change value by means other than the current program
`const`	fixed value, defined in the source code and can not be changed during execution
`unsigned`	range starts with 0 includes only positive values
`signed`	range includes both negative and positive values

Table 4-3 Variable modifiers

Refer to the Appendix 1: C Declarations - A Short Primer for a brief Primer on how to read and write declarations in C.

In all cases **const** means the variable has a fixed value and cannot be changed. When defining a constant global on an embedded system like the Cortex M, the parameter will be allocated in ROM. In the following example, **Ret** is allocated in ROM. When **const** is added to a parameter or a local variable, it means that parameter can not be modified by the function. It does not change where the parameter is allocated. For example, this example is legal.

```
unsigned char const Ret=13;
void LegalFuntion(short in){
  while(in){
    UART_OutChar(Ret);
    in--;
  }
}
```

On the other hand, this example is not legal because the function attempts to modify the input parameter. **`in`** in this example would have been allocated on the stack or in a register.

```
void NotLegalFuntion(const short in){
  while(in){
    UART_OutChar(13);
    in--;  // this operation is illegal
  }
}
```

Similarly, this example is not legal because the function attempts to modify the local variable. **`count`** in this example would have been allocated on the stack or in a register.

```
void NotLegalFuntion2(void){ const short count=5;
  while(count){
    UART_OutChar(13);
    count--;  // this operation is illegal
  }
}
```

As we shall see, a similar syntax is used to declare pointers, arrays, and functions (Chapters 7, 8, and 10).

Character Variables

Character variables are stored as 8-bit quantities. When they are fetched from memory, they are always promoted automatically to 32-bit integers. Unsigned 8-bit values are promoted by adding 24 zeros into the most significant bits. Signed values are promoted by coping the sign bit (bit7) into the 24 most significant bits.

There is a confusion when signed and unsigned variables are mixed into the same expression. It is good programming practice to avoid such confusions. As with integers, when a signed character enters into an operation with an unsigned quantity, the character is interpreted as though it was unsigned. The result of such operations is also unsigned. When a signed character joins with another signed quantity, the result is also signed.

```
char x;   /* signed 8 bit global */
unsigned short y;   /* unsigned signed 16 bit global */
void sub(void){
    y=y+x;
/* x treated as unsigned even though defined as signed */
}
```

Listing 4-13: An example showing the mixture of signed and unsigned variables

There is also a need to change the size of characters when they are stored, since they are represented in the CPU as 32-bit values. In this case, however, it matters not whether they are signed or unsigned. Obviously there is only one reasonable way to put a 32-bit quantity into an 8-bit location. When the high-order 24 bits are chopped off, an error might occur. It is the programmer's responsibility to ensure that significant bits are not lost when characters are stored.

When Do We Use Automatics Versus Statics?

Because their contents are allowed to change, all variables must be allocated in RAM and not ROM. An **automatic variable** contains temporary information used only by one software module. As we saw, automatic variables are typically allocated, used, then deallocated from the stack. Since an interrupt will save registers and create its own stack frame, the use of automatic variables is important for creating reentrant software. Automatic variables provide protection limiting the scope of access in such a way that only the program that created the local variable can access it. The information stored in an automatic variable is not permanent. This means if we store a value into an automatic variable during one execution of the module, the next time that module is executed the previous value is not available. Typically we use automatics for loop counters, temporary sums. We use an automatic variable to store data that is temporary in nature. In summary, reasons why we place automatic variables on the stack include

• dynamic allocation release allows for reuse of memory
• limited scope of access provides for data protection
• can be made reentrant.
• limited scope of access provides for data protection
• since absolute addressing is not used, the code is relocatable
• the number of variables is only limited by the size of the stack allocation.

A **static variable** is information shared by more than one program module. E.g., we use globals to pass data between the main (or foreground) process and an interrupt (or background) process. Static variables are not deallocated. The information they store is permanent. We can use static variables for the time of day, date, user name, temperature, pointers to shared data. The Metrowerks compiler uses absolute addressing (direct or extended) to access the static variables.

Initialization of variables and constants

Most programming languages provide ways of specifying *initial values*; that is, the values that variables have when program execution begins. We saw earlier that the Metrowerks compiler will initially set all static variables to zero. Constants must be initialized at the time they are declared, and we have the option of initializing the variables.

Specifying initial values is simple. In its declaration, we follow a variable's name with an equal sign and a constant expression for the desired value. Thus

```
short Temperature = -55;
```

declares `Temperature` to be a 16-bit signed integer, and gives it an initial value of -55. Character constants with backslash-escape sequences are permitted. Thus

```
char Letter = '\t';
```

declares `Letter` to be a character, and gives it the value of the tab character. If array elements are being initialized, a list of constant expressions, separated by commas and enclosed in braces, is written. For example,

```
const unsigned short Steps[4] = {10, 9, 6, 5};
```

declares `Steps` to be an unsigned 16-bit constant integer array, and gives its elements the values 10, 9, 6, and 5 respectively. If the size of the array is not specified, it is determined by the number of initializers. Thus

```
char Waveform[] = {28,27,60,30,40,50,60};
```

declares `Waveform` to be a signed 8-bit array of 7 elements which are initialized to the `28,27,60,30,40,50,60`. On the other hand, if the size of the array is given and if it exceeds the number of initializers, the leading elements are initialized and the trailing elements default to zero. Therefore,

```
char Waveform[100] = {28,27,60,30,40,50,60};
```

declares `Waveform` to be an integer array of 100 elements, the first 7 elements of which are initialized to the `28,27,60,30,40,50,60` and the others to zero. Finally, if the size of an array is given and there are too many initializers, the compiler generates an error message. In that case, the programmer must be confused.

Character arrays and character pointers may be initialized with a character string. In these cases, a terminating zero is automatically generated. For example,

```
char Name[4] = "Jon";
```

declares `Name` to be a character array of four elements with the first three initialized to 'J', 'o', and 'n' respectively. The fourth element contains zero. If the size of the array is not given, it will be set to the size of the string plus one. Thus ca in

```
char Name[] = "Jon";
```

also contains the same four elements. If the size is given and the string is shorter, trailing elements default to zero. For example, the array declared by

```
char Name[6] = "Jon";
```

contains zeroes in its last three elements. If the string is longer than the specified size of the array, the array size is increased to match. If we write

```
char *NamePt = "Jon";
```

the effect is quite different from initializing an array. First a word (16 bits) is set aside for the pointer itself. This pointer is then given the address of the string. Then, beginning with that byte, the string and its zero terminator are assembled. The result is that `NamePt` contains the address of the string "Jon". The Imagecraft and Metrowerks compilers accept initializers for character variables, pointers, and arrays, and for integer variables and arrays. The initializers themselves may be either constant expressions, lists of constant expressions, or strings.

Implementation of the initialization

The compiler initializes static constants simply by defining its value in ROM. In the following example, J is a static constant (actually K is a literal)

```
short I;            /* 16 bit global */
const short J=96;   /* 16 bit constant */
#define K 97;
void main(void){
    I=J;
    I=K;}
```

Listing 4-14: An example showing the initialization of a static constant

Even though the following two applications of global variable are technically proper, the explicit initialization of global variables in my opinion is a better style.

```
/* poor style */      /* good style */
int I=95;             int I;
void main(void){      void main(void){
                          I=95;
}                     }
```

Opinion: I firmly believe a good understanding of the assembly code generated by our compiler makes us better programmers.

Go to Chapter 5 on Expressions Return to Table of Contents

Chapter 5: Expressions

What's in Chapter 5?

Most programming languages support the traditional concept of an expression as a combination of constants, variables, array elements, and function calls joined by various operators (+, -, etc.) to produce a single numeric value. Each operator is applied to one or two operands (the values operated on) to produce a single value which may itself be an operand for another operator. This idea is generalized in C by including nontraditional data types and a rich set of operators. Pointers, unsubscripted array names, and function names are allowed as operands. And, as Tables 5-1 through 5-6 illustrate, many operators are available. All of these operators can be combined in any useful manner in an expression. As a result, C allows the writing very compact and efficient expressions which at first glance may seem a bit strange. Another unusual feature of C is that anywhere the syntax calls for an expression, a list of expressions, with comma separators, may appear.

Precedence and associativity

The basic problem in evaluating expressions is deciding which parts of an expression are to be associated with which operators. To eliminate ambiguity, operators are given three properties: *operand count*, *precedence*, and *associativity*.

Operand count refers to the classification of operators as unary, binary, or ternary according to whether they operate on one, two, or three operands. The unary minus sign, for instance, reverses the sign of the following operand, whereas the binary minus sign subtracts one operand from another.

The following example converts the distance x in inches to a distance y in cm. Without parentheses the following statement seems ambiguous

```
y = 254*x/100;
```

If we divide first, then y can only take on values that are multiples of 254 (e.g., 0 254 508 etc.) So the following statement is incorrect.

```
y = 254*(x/100);
```

The proper approach is to multiply first then divide. To multiply first we must guarantee that the product **254*x** will not overflow the precision of the computer. How do we know what precision the compiler used for the intermediate result **254*x**? To answer this question, we must observe the assembly code generated by the compiler. Since multiplication and division associate left to right, the first statement without parentheses although ambiguous will actually calculate the correct answer. It is good programming style to use parentheses to clarify the expression. So this last statement has both good style and proper calculation.

```
y = (254*x)/100;
```

The issues of precedence and associativity were explained in Chapter 1. Precedence defines the evaluation order. For example the expression 3+4*2 will be 11 because multiplication as precedence over addition. Associativity determines the order of execution for operators that have the same precedence. For example, the expression 10-3-2 will be 5, because subtraction associates left to right. On the other hand, if x and y are initially 10, then the expression x+=y+=1 will first make y=y+1 (11), then make x=x+y (21) because the operator += associates right to left. The table from chapter 1 is repeated for your convenience.

Precedence	Operators	Associativity
highest	() [] . -> ++(postfix) --(postfix)	left to right
	++(prefix) --(prefix) !~ **sizeof**(type) +(unary) -(unary) &(address) *(dereference)	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= <<= >>= |= &= ^=	right to left
lowest	,	left to right

Table 1-4: Precedence and associativity determine the order of operation

Unary operators

We begin with the unary operators, which take a single input and give a single output. In the following examples, assume all numbers are 16 bit signed (short). The following variables are listed

```
short data; /* -32767 to +32767 */
short *pt;  /* pointer to memory */
short flag; /* 0 is false, not zero is true */
```

`operator`	`meaning`	*example*	*result*
`~`	binary complement	~0x1234	0xEDCB
`!`	logical complement	!flag	flip 0 to 1 and notzero to 0
`&`	address of	&data	address in memory where data is stored
`-`	negate	-100	negative 100
`+`	positive	+100	100
`++`	preincrement	++data	data=data+1, then result is data
`--`	predecrement	--data	data=data-1, then result is data

*	reference	*pt	16 bit information pointed to by pt

Table 5.1: Unary prefix operators.

operator	*meaning*	*example*	*result*
++	postincrement	data++	result is data, then data=data+1
--	postdecrement	data--	result is data, then data=data+1

Table 5.2: Unary postfix operators.

Binary operators

Next we list the binary arithmetic operators, which operate on two number inputs giving a single number result. The operations of addition, subtraction and shift left are the same independent of whether the numbers are signed or unsigned. As we will see later, overflow and underflow after an addition, subtraction and shift left are different for signed and unsigned numbers, but the operation itself is the same. On the other hand multiplication, division, and shift right have different functions depending on whether the numbers are signed or unsigned. It will be important, therefore, to avoid multiplying or dividing an unsigned number with a signed number.

operator	*meaning*	*example*	*result*
+	addition	100+300	400
-	subtraction	100-300	-200
*	multiplication	10*300	3000
/	division	123/10	12
%	remainder	123%10	3
<<	shift left	102<<2	408
>>	shift right	102>>2	25

Table 5.3: Binary arithmetic operators.

The binary bitwise logical operators take two inputs and give a single result.

operator	*meaning*	*example*	*result*
&	bitwise and	0x1234&0x00FF	0x0034
|	bitwise or	0x1234|0x00FF	0x12FF
^	bitwise exclusive or	0x1234^0x00FF	0x12CB

Table 5.4: Binary bitwise logical operators.

The binary Boolean operators take two Boolean inputs and give a single Boolean result.

operator	*meaning*	*example*	*result*
&&	and	0 && 1	0 (false)
||	or	0 || 1	1 (true)

Table 5.5: Binary Boolean operators.

Many programmers confuse the logical operators with the Boolean operators. Logical operators take two numbers and perform a bitwise logical operation. Boolean operators take two Boolean inputs (0 and notzero) and return a Boolean (0 or 1). In the program below, the operation `c=a&b;` will perform a *bitwise logical and* of 0x0F0F **and** 0xF0F0 resulting in 0x0000. In the `d=a&&b;` expression, the value a is considered as a true (because it is not zero) and the value b also is considered a true (not zero). The Boolean operation of true **and** true gives a true result (1).

```
short a,b,c,d;
int main(void){ a=0x0F0F; b=F0F0;
  c = a&b;  /* logical result c will be 0x0000 */
  d = a&&b; /* Boolean result d will be 1 (true) */
  return 1;
}
```

Listing 5-1: Illustration of the difference between logical and Boolean operators

The binary relational operators take two number inputs and give a single Boolean result.

operator	*meaning*	*example*	*result*
==	equal	100 == 200	0 (false)
!=	not equal	100 != 200	1 (true)
<	less than	100 < 200	1 (true)
<=	less than or equal	100 <= 200	1 (true)
>	greater than	100 > 200	0 (false)
>=	greater than or equal	100 >= 200	0 (false)

Table 5.6: Binary relational operators.

Some programmers confuse *assignment equals* with the *relational equals*. In the following example, the first **if** will execute the `subfunction()` if a is equal to zero (a is not modified). In the second case, the variable b is set to zero, and the `subfunction()` will never be executed because the result of the equals assignment is the value (in this case the 0 means false).

```
short a,b;
void program(void){
  if(a==0) subfunction(); /* execute subfunction if a is zero */
  if(b=0) subfunction();  /* set b to zero, never execute subfunction */
}
```

Listing 5-2: Illustration of the difference between relational and assignment equals

Before looking at the kinds of expressions we can write in C, we will first consider the process of evaluating expressions and some general properties of operators.

Assignment Operators

The assignment operator is used to store data into variables. The syntax is `variable=expression;` where `variable` has been previously defined. At run time, the result of the expression is saved into the variable. If the type of the expression is different from the variable, then the result is automatically converted. For more information about types and conversion, see expression type and explicit casting. The assignment operation itself has a result, so the assignment operation can be nested.

```
short a,b;
void initialize(void){
  a = b = 0; /* set both variables to zero */
}
```

Listing 5-3: Example of a nested assignment operation

The read/modify write assignment operators are convenient. Examples are shown below.

```
short a,b;
void initialize(void){
    a += b;  /* same as a=a+b */
    a -= b;  /* same as a=a-b */
    a *= b;  /* same as a=a*b */
    a /= b;  /* same as a=a/b */
    a %= b;  /* same as a=a%b */
    a <<= b; /* same as a=a<<b */
    a <<= b; /* same as a=a<<b */
    a >>= b; /* same as a=a>>b */
    a |= b;  /* same as a=a|b */
    a &= b;  /* same as a=a&b */
    a ^= b;  /* same as a=a^b */
}
```

Listing 5-4 List of all read/modify/write assignment operations

Most compilers will produce the same code for the short and long version of the operation. Therefore you should use the read/modify/write operations only in situations that make the software easier to understand.

```
void function(void){
  PORTA |= 0x01;  /* set PA0 high */
  PORTB &=~ 0x80; /* clear PB7 low */
  PORTC ^= 0x40;  /* toggle PC6 */
}
```

Listing 5-5 Good examples of read/modify/write assignment operations

Expression Types and Explicit Casting

We saw earlier that numbers are represented in the computer using a wide range of formats. A list of these formats is given in Table 5.7. Notice that for the Cortex M, the **int** and **long** types are the same. On the other hand with the Freescale 9S12, the **int** and **short** types are the same. This difference may cause confusion, when porting code from one system to another. I suggest you use the **int** type when you are interested in efficiency and don't care about precision, and use the **short** type when you want a variable with a 16-bit precision.

type	range	precision	example variable
unsigned char	0 to 255	8 bits	unsigned char uc;
char	-128 to 127	8 bits	char sc;
unsigned int	0 to 4294967295	32 bits	unsigned int ui;
int	-2147483648 to 2147483647	32 bits	int si;
unsigned short	0 to 65535U	16 bits	unsigned short us;
short	-32768 to 32767	16 bits	short ss;
long	-2147483648 to 2147483647	32 bits	long sl;
unsigned long	0 to 4294967295	32 bits	unsigned long ui;

Table 5-7. Available number formats for the compiler

An obvious question arises, what happens when two numbers of different types are operated on? Before operation, the C compiler will first convert one or both numbers so they have the same type. The conversion of one type into another has many names:
  automatic conversion,
  implicit conversion,
  coercion,
  promotion, or
  widening.

There are three ways to consider this issue. The first way to think about this is if the range of one type completely fits within the range of the other, then the number with the smaller range is converted (promoted) to the type of the number with the larger range. In the following examples, a number of type1 is added to a number of type2. In each case, the number range of type1 fits into the range of type2, so the parameter of type1 is first promoted to type2 before the addition.

type1		type2	example
unsigned char	fits inside	unsigned short	uc+us is of type unsigned short
unsigned char	fits inside	short	uc+ss is of type short
unsigned char	fits inside	long	uc+sl is of type long
char	fits inside	short	sc+ss is of type short
char	fits inside	long	sc+sl is of type long
unsigned short	fits inside	long	us+sl is of type long
short	fits inside	long	ss+sl is of type long

Table 5-8. When the range of one type fits inside the range of another, then conversion is simple

The second way to consider mixed precision operations is that in most cases the compiler will promote the number with the smaller precision into the other type before operation. If the two numbers are of the same precision, then the signed number is converted to unsigned. These automatic conversions may not yield correct results. The third and best way to deal with mixed type operations is to perform the conversions explicitly using the **cast** operation. We can force the type of an expression by explicitly defining its type. This approach allows the programmer to explicitly choose the type of the operation. Consider the following digital filter with mixed type operations. In this example, we explicitly convert x and y to signed 16 bit numbers and perform 16 bit signed arithmetic. Note that the assignment of the result into y, will require a demotion of the 16 bit signed number into 8 bit signed. Unfortunately, C does not provide any simple mechanisms for error detection/correction (see overflow and underflow.)

```
char y; // output of the filter
unsigned char x; // input of the filter
void filter(void){
  y = (12*(short)x + 56*(short)y)/100;
}
```

Listing 5-6: Examples of the selection operator

We apply an explicit cast simply by preceding the number or expression with parentheses surrounding the type. In this next digital filter all numbers are of the same type. Even so, we are worried that the intermediate result of the multiplications and additions might overflow the 16-bit arithmetic. We know from digital signal processing that the final result will always fit into the 16-bit variable. For more information on the design and analysis of digital filters, see Chapter 5 of Embedded Systems: Real-Time Operating Systems for ARM Cortex M Microcontrollers by Jonathan W. Valvano. In this example, the cast `(long)` will specify the calculations be performed in 32-bit precision.

```
// y(n) = [113*x(n) + 113*x(n-2) - 98*y(n-2)]/128, channel specifies the A/D channel
short x[3],y[3]; // MACQs containing current and previous
void SysTick_Handler(void){
  y[2]=y[1]; y[1]=y[0]; // shift MACQ
  x[2]=x[1]; x[1]=x[0];
  x[0] = A2D(channel); // new data
  y[0] = (113*((long)x[0]+(long)x[2])-98*(long)y[2])>>7;}
```

Listing 5-7: We can use a cast to force higher precision arithmetic

We saw in Chapter 1, casting was used to assign a symbolic name to an I/O port. In particular the following define casts the number `0x400043FC` as a pointer type, which points to an unsigned 32-bit data. More about pointers can be found in Chapter 7.

```
#define GPIO_PORTA_DATA_R (*((volatile unsigned long *)0x400043FC))
```

Selection operator

The selection operator takes three input parameters and yields one output result. The format is

Expr1 ? Expr2 : Expr3

The first input parameter is an expression, Expr1, which yields a boolean (0 for false, not zero for true). Expr2 and Expr3 return values that are regular numbers. The selection operator will return the result of Expr2 if the value of Expr1 is true, and will return the result of Expr3 if the value of Expr1 is false. The type of the expression is determined by the types of Expr2 and Expr3. If Expr2 and Expr3 have different types, then the usual promotion is applied. The resulting time is determined at compile time, in a similar manner as the Expr2+Expr3 operation, and not at run time depending on the value of Expr1. The following two subroutines have identical functions.

```
short a,b;
void sub1(void){
  a = (b==1) ? 10 : 1;
}
void sub2(void){
  if(b==1)
     a=10;
  else
     a=1;
}
```

Listing 5-8: Examples of the selection operator

Arithmetic Overflow and Underflow

An important issue when performing arithmetic calculations on integer values is the problem of underflow and overflow. Arithmetic operations include addition, subtraction, multiplication, division and shifting. Overflow and underflow errors can occur during all of these operations. In assembly language the programmer is warned that an error has occurred because the processor will set condition code bits after each of these operations. Unfortunately, the C compiler provides no direct access to these error codes, so we must develop careful strategies for dealing with overflow and underflow. It is important to remember that arithmetic operations (addition, subtraction, multiplication, division, and shifting) have constraints when performed with finite precision on a microcomputer. An overflow error occurs when the result of an arithmetic operation can not fit into the finite precision of the result. We will study addition and subtraction operations in detail, but the techniques for dealing with overflow and underflow will apply to the other arithmetic operations as well. We will consider two approaches

avoiding the error
detecting the error then correcting the result

For example when two 8 bit numbers are added, the sum may not fit back into the 8 bit result. We saw earlier that the same digital hardware (instructions) could be used to add and subtract unsigned and signed numbers. Unfortunately, we will have to design separate overflow detection for signed and unsigned addition and subtraction.

All microcomputers have a condition code register which contain bits which specify the status of the most recent operation. In this section, we will introduce 4 condition code bits common to most microcomputers. If the two inputs to an addition or subtraction operation are considered as unsigned, then the C bit (carry) will be set if the result does not fit. In other words, after an unsigned addition, the C bit is set if the answer is wrong. If the two inputs to an addition or subtraction operation are considered as signed, then the V bit (overflow) will be set if the result does not fit. In other words, after a signed addition, the V bit is set if the answer is wrong. The Freescale 6805 does not have a V bit, therefore it will be difficult to check for errors after an operation on signed numbers.

bit	name	meaning after addition or subtraction
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow

C	carry	unsigned overflow

Table 5.9. Condition code bits contain the status of the previous arithmetic or logical operation.

For an 8 bit unsigned number, there are only 256 possible values, 0 to 255. We can think of the numbers as positions along a circle. There is a discontinuity at the 0|255 interface, everywhere else adjacent numbers differ by &plusmn;1. If we add two unsigned numbers, we start at the position of the first number a move in a clockwise direction the number of steps equal to the second number. For example, if 96+64 is performed in 8 bit unsigned precision, the correct result of 160 is obtained. In this case, the carry bit will be 0 signifying the answer is correct. On the other hand, if 224+64 is performed in 8 bit unsigned precision, the incorrect result of 32 is obtained. In this case, the carry bit will be 1, signifying the answer is wrong.

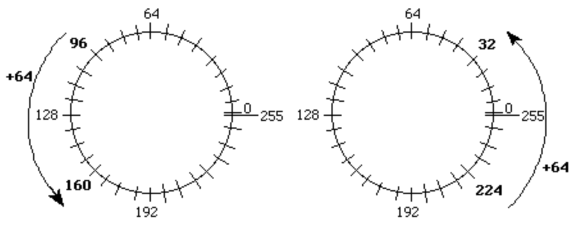


Figure 5-1: 8 bit unsigned addition.

For subtraction, we start at the position of the first number a move in a counterclockwise direction the number of steps equal to the second number. For example, if 160-64 is performed in 8 bit unsigned precision, the correct result of 96 is obtained (carry bit will be 0.) On the other hand, if 32-64 is performed in 8 bit unsigned precision, the incorrect result of 224 is obtained (carry bit will be 1.)

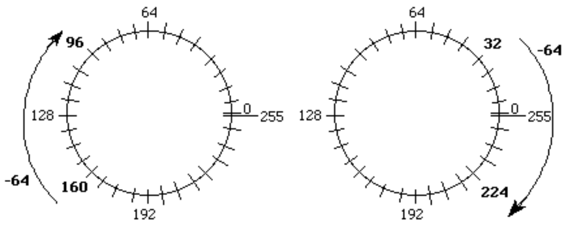


Figure 5-2: 8 bit unsigned subtraction.

In general, we see that the carry bit is set when we cross over from 255 to 0 while adding or cross over from 0 to 255 while subtracting.

Observation: The carry bit, C, is set after an unsigned add or subtract when the result is incorrect.

For an 8 bit signed number, the possible values range from -128 to 127. Again there is a discontinuity, but this time it exists at the -128|127 interface, everywhere else adjacent numbers differ by &plusmn;1. The meanings of the numbers with bit 7=1 are different from unsigned, but we add and subtract signed numbers on the number wheel in a similar way (e.g., addition of a positive number moves clockwise.) Adding a negative number is the same as subtracting a positive number hence this operation would cause a counterclockwise motion. For example, if -32+64 is performed, the correct result of 32 is obtained. In this case, the overflow bit will be 0 signifying the answer is correct. On the other hand, if 96+64 is performed, the incorrect result of -96 is obtained. In this case, the overflow bit will be 1 signifying the answer is wrong.

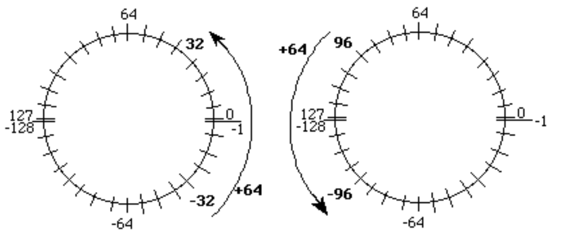


Figure 5-3: 8 bit signed addition.

For subtracting signed numbers, we again move in a counterclockwise direction. Subtracting a negative number is the same as adding a positive number hence this operation would cause a clockwise motion. For example, if 32-64 is performed, the correct result of -32 is obtained (overflow bit will be 0.) On the other hand, if -96-64 is performed, the incorrect result of 96 is obtained (overflow bit will be 1.)

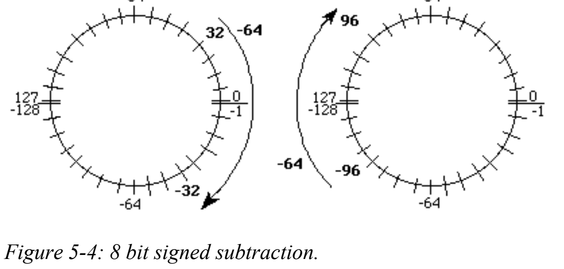


Figure 5-4: 8 bit signed subtraction.

In general, we see that the overflow bit is set when we cross over from 127 to -128 while adding or cross over from -128 to 127 while subtracting.

Observation: The overflow bit, V, is set after a signed add or subtract when the result is incorrect.

Another way to determine the overflow bit after an addition is to consider the carry out of bit 6. The V bit will be set of there is a carry out of bit 6 (into bit 7) but no carry out of bit 7 (into the C bit). It is also set if there is no carry out of bit 6 but there is a carry out of bit 7. Let X7,X6,X5,X4,X3,X2,X1,X0 and

M7,M6,M5,M4,M3,M2,M1,M0 be the individual binary bits of the two 8 bit numbers which are to be added, and let R7,R6,R5,R4,R3,R2,R1,R0 be individual binary bits of the 8 bit sum. Then, the 4 condition code bits after an addition are shown in Table 5.10.

N	R7	if unsigned result above 127, if signed result is negative
Z	$\overline{R7}\cdot\overline{R6}\cdot\overline{R5}\cdot\overline{R4}\cdot\overline{R3}\cdot\overline{R2}\cdot\overline{R1}\cdot\overline{R0}$	result is zero
V	$X7\cdot M7\cdot\overline{R7}$	add two positives got a negative result
	$+\overline{X7}\cdot\overline{M7}\cdot R7$	or add two negatives got a positive result
C	$X7\cdot M7$	add two numbers both above 127
	$+M7\cdot\overline{R7}$	or add one number above 127 and got a number below 128
	$+X7\cdot\overline{R7}$	or add one number above 127 and got a number below 128

Table 5.10. Condition code bits after an 8 bit addition operation.

Let the result R be the result of the subtraction X-M. Then, the 4 condition code bits are shown in Table 5.11.

N	R7	if unsigned result above 127, if signed result is negative
Z	$\overline{R7}\cdot\overline{R6}\cdot\overline{R5}\cdot\overline{R4}\cdot\overline{R3}\cdot\overline{R2}\cdot\overline{R1}\cdot\overline{R0}$	result is zero
V	$X7\cdot\overline{M7}\cdot\overline{R7}$	a positive minus a negative and got a positive result
	$+\overline{X7}\cdot M7\cdot R7$	or a negative minus a positive and got a negative result
C	$\overline{X7}\cdot M7$	a number below 128 minus a number above 127
	$+M7\cdot R7$	or subtracted a number above 127 and got one above 127
	$+\overline{X7}\cdot R7$	or started with a number below 127 and got one above 127

Table 5-11. Condition code bits after an 8 bit subtraction operation.

Common Error: Ignoring overflow (signed or unsigned) can result in significant errors.

Observation: Microcomputers have two sets of conditional branch instructions (if statements) which make program decisions based on either the C or V bit.

Common Error: An error will occur if you unsigned conditional branch instructions (if statements) after operating on signed numbers, and vice-versa.

There are some applications where arithmetic errors are not possible. For example if we had two 8 bit unsigned numbers that we knew were in the range of 0 to 100, then no overflow is possible when they are added together.

Typically the numbers we are processing are either signed or unsigned (but not both), so we need only consider the corresponding C or V bit (but not both the C and V bits at the same time.) In other words, if the two numbers are unsigned, then we look at the C bit and ignore the V bit. Conversely, if the two numbers are signed, then we look at the V bit and ignore the C bit. There are two appropriate mechanisms to deal with the potential for arithmetic errors when adding and subtracting. The first mechanism, used by most compilers, is called promotion. Promotion involves increasing the precision of the input numbers, and performing the operation at that higher precision. An error can still occur if the result is stored back into the smaller precision. Fortunately, the program has the ability to test the intermediate result to see if it will fit into the smaller precision. To promote an unsigned number we add zero's to the left side. In a previous example, we added the unsigned 8 bit 224 to 64, and got the wrong result of 32. With promotion we first convert the two 8 bit numbers to 16 bits, then add.

We can check the 16 bit intermediate result (e.g., 228) to see if the answer will fit back into the 8 bit result. In the following flowchart, X and M are 8 bit unsigned inputs, X16, M16, and R16 are 16 bit intermediate values, and R is an 8 bit unsigned output. The oval symbol represents the entry and exit points, the rectangle is used for calculations, and the diamond shows a decision. Later in the book we will use parallelograms and trapezoids to perform input/output functions.

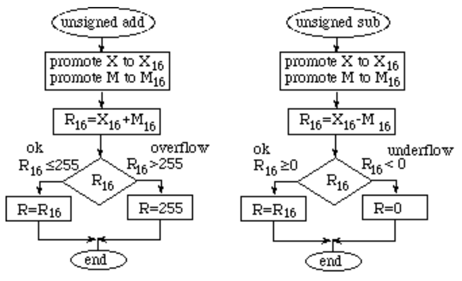


Figure 5-5: Promotion can be used to avoid overflow and underflow.

To promote a signed number, we duplicate the sign bit as we add binary digits to the left side. Earlier, we performed the 8 bit signed operation -96-64 and got a signed overflow. With promotion we first convert the two numbers to 16 bits, then subtract.

```
decimal      8 bit         16 bit
  -96      1010,0000   1111,1111,1010,0000
 - 64     -0100,0000  -0000,0000,0100,0000
 -160      0110,0000   1111,1111,0110,0000
```

We can check the 16 bit intermediate result (e.g., -160) to see if the answer will fit back into the 8 bit result. In the following flowchart, X and M are 8 bit signed inputs, X16, M16, and R16 are 16 bit signed intermediate values, and R is an 8 bit signed output.

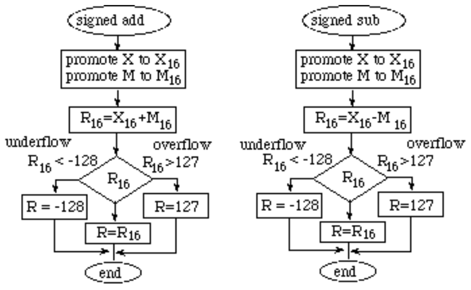


Figure 5-6: Promotion can be used to avoid overflow and underflow.

The other mechanism for handling addition and subtraction errors is called ceiling and floor. It is analogous to movements inside a room. If we try to move up (add a positive number or subtract a negative number) the ceiling will prevent us from exceeding the bounds of the room. Similarly, if we try to move down (subtract a positive number or add a negative number) the floor will prevent us from going too low. For our 8 bit addition and subtraction, we will prevent the 0 to 255 and 255 to 0 crossovers for unsigned operations and -128 to +127 and +127 to -128 crossovers for signed operations. These operations are described by the following flowcharts. If the carry bit is set after an unsigned addition the result is adjusted to the largest possible unsigned number (ceiling). If the carry bit is set after an unsigned subtraction, the result is adjusted to the smallest possible unsigned number (floor.)

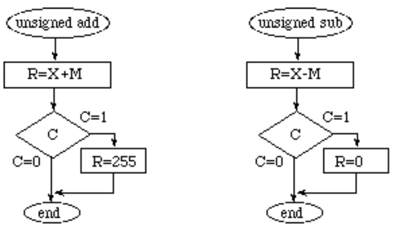


Figure 5-7: In assembly language we can detect overflow and underflow.

If the overflow bit is set after a signed operation the result is adjusted to the largest (ceiling) or smallest (floor) possible signed number depending on whether it was a -128 to 127 cross over (N=0) or 127 to -128 cross over (N=1). Notice that after a signed overflow, bit 7 of the result is always wrong because there was a cross over.

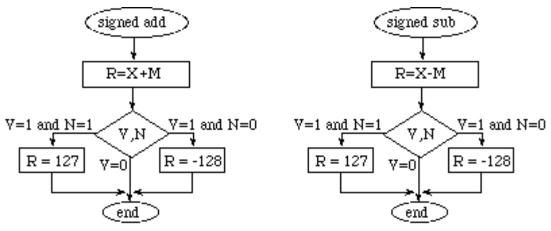


Figure 5-8: In assembly language we can detect overflow and underflow.

Go to Chapter 6 on Statements Return to Table of Contents

Chapter 6: Flow of Control

What's in Chapter 6?

Every procedural language provides statements for determining the flow of control within programs. Although declarations are a type of statement, in C the unqualified word statement usually refers to procedural statements rather than declarations. In this chapter we are concerned only with procedural statements.

In the C language, statements can be written only within the body of a function; more specifically, only within compound statements. The normal flow of control among statements is sequential, proceeding from one statement to the next. However, as we shall see, most of the statements in C are designed to alter this sequential flow so that algorithms of arbitrary complexity can be implemented. This is done with statements that control whether or not other statements execute and, if so, how many times. Furthermore, the ability to write compound statements permits the writing a sequence of statements wherever a single, possibly controlled, statement is allowed. These two features provide the necessary generality to implement any algorithm, and to do it in a structured way.

Simple Statements

The C language uses semicolons as statement terminators. A semicolon follows every simple (non-compound) statement, even the last one in a sequence.

When one statement controls other statements, a terminator is applied only to the controlled statements. Thus we would write

```
if(x > 5) x = 0; else ++x;
```

with two semicolons, not three. Perhaps one good way to remember this is to think of statements that control other statements as "super" statements that "contain" ordinary (simple and compound) statements. Then remember that only simple statements are terminated. This implies, as stated above, that compound statements are not terminated with semicolons. Thus

```
while(x < 5) {func(); ++x;}
```

is perfectly correct. Notice that each of the simple statements within the compound statement is terminated.

Compound Statements

The terms compound statement and block both refer to a collection of statements that are enclosed in braces to form a single unit. Compound statements have the form

```
{ObjectDeclaration?... Statement?... }
```

ObjectDeclaration?... is an optional set of local declarations. If present, C requires that they precede the statements; in other words, they must be written at the head of the block. Statement?... is a series of zero or more simple or compound statements. Notice that there is not a semicolon at the end of a block; the closing brace suffices to delimit the end. In this example the local variable `temp` is only defined within the inner compound statement.

```
int main(void){ short n1,n2;
  n1=1; n2=2;
  { short temp;
    temp=n1; n1=n2; n2=temp; /* switch n1,n2 */
  }
  return 1;
}
```

Listing 6.1: Examples of a compound statements

The power of compound statements derives from the fact that one may be placed anywhere the syntax calls for a statement. Thus any statement that controls other statements is able to control units of logic of any complexity.

When control passes into a compound statement, two things happen. First, space is reserved on the stack for the storage of local variables that are declared at the head of the block. Then the executable statements are processed.

One important limitation in C is that a block containing local declarations must be entered through its leading brace. This is because bypassing the head of a block effectively skips the logic that reserves space for local objects. Since the goto and switch statements (below) could violate this rule.

The If Statement

If statements provide a non-iterative choice between alternate paths based on specified conditions. They have either of two forms

```
if ( ExpressionList ) Statement1
```

or

```
if ( ExpressionList ) Statement1
else Statement2
```

ExpressionList is a list of one or more expressions and Statement is any simple or compound statement. First, ExpressionList is evaluated and tested. If more than one expression is given, they are evaluated from left to right and the right-most expression is tested. If the result is true (non-zero), then the Statement1 is executed and the Statement2 (if present) is skipped. If it is false (zero), then Statement1 is skipped and Statement2 (if present) is executed. In this first example, the function `isGreater()` is executed if G2 is larger than 100.

```
if(G2 > 100) isGreater();
```

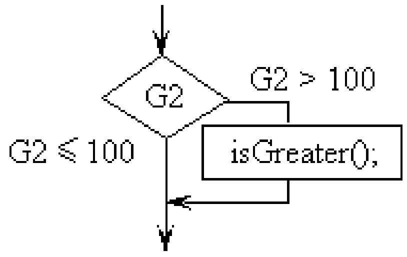


Figure 6.1: Example if statement.

A 3-wide median filter can be designed using if-else conditional statements.

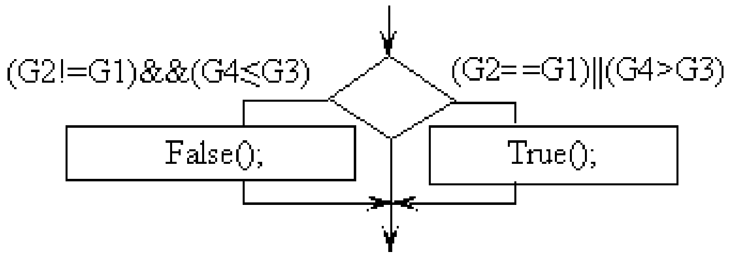
```
short Median(short u1,short u2,short u3){ short result;
  if(u1>u2)
    if(u2>u3)    result=u2;   // u1>u2,u2>u3       u1>u2>u3
    else
       if(u1>u3) result=u3;   // u1>u2,u3>u2,u1>u3 u1>u3>u2
       else      result=u1;   // u1>u2,u3>u2,u3>u1 u3>u1>u2
  else
    if(u3>u2)    result=u2;   // u2>u1,u3>u2       u3>u2>u1
    else
       if(u1>u3) result=u1;   // u2>u1,u2>u3,u1>u3 u2>u1>u3
       else      result=u3;   // u2>u1,u2>u3,u3>u1 u2>u3>u1
  return(result):}
```

Listing 6.2: A 3-wide median function.

For more information on the design and analysis of digital filters, see Chapter 5 of Embedded Systems: Real-Time Operating Systems for ARM Cortex M Microcontrollers by Jonathan W. Valvano

Complex conditional testing can be implemented using the relational and Boolean operators described in the last chapter.

```
if ((G2==G1)||(G4>G3)) True(); else False();
```



The Switch Statement

Switch statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. Switch statements have the form

```
switch ( ExpressionList ) { Statement?...}
```

where ExpressionList is a list of one or more expressions. Statement?... represents the statements to be selected for execution. They are selected by means of case and default prefixes--special labels that are used only within switch statements. These prefixes locate points to which control jumps depending on the value of ExpressionList. They are to the switch statement what ordinary labels are to the goto statement. They may occur only within the braces that delimit the body of a switch statement.

The case prefix has the form

```
case ConstantExpression :
```

and the default prefix has the form

```
default:
```

The terminating colons are required; they heighten the analogy to ordinary statement labels. Any expression involving only numeric and character constants and operators is valid in the case prefix.

After evaluating ExpressionList, a search is made for the first matching case prefix. Control then goes directly to that point and proceeds normally from there. Other case prefixes and the default prefix have no effect once a case has been selected; control flows through them just as though they were not even there. If no matching case is found, control goes to the default prefix, if there is one. In the absence of a default prefix, the entire compound statement is ignored and control resumes with whatever follows the switch statement. Only one default prefix may be used with each switch.

If it is not desirable to have control proceed from the selected prefix all the way to the end of the switch block, break statements may be used to exit the block. Break statements have the form

```
break;
```

Some examples may help clarify these ideas. Assume Port A is specified as an output, and bits 3,2,1,0 are connected to a stepper motor. The switch statement will first read Port A and the data with 0x0F `(GPIO_PORTA_DATA_R&0x0F)`. If the result is 5, then PortA is set to 6 and control is passed to the end of the switch (because of the break). Similarly for the other 3 possibilities

```
#define GPIO_PORTA_DATA_R       (*((volatile unsigned long *)0x400043FC))
void step(void){ /* turn stepper motor one step */
   switch (GPIO_PORTA_DATA_R&0x0F) {
      case 0x05:
         GPIO_PORTA_DATA_R=0x06; // 6 follows 5;
         break;
      case 0x06:
         GPIO_PORTA_DATA_R=0x0A; // 10 follows 6;
         break;
      case 0x0A:
         GPIO_PORTA_DATA_R=0x09; // 9 follows 10;
         break;
      case 0x09:
         GPIO_PORTA_DATA_R=0x05; // 5 follows 9;
         break;
      default:
         PORTA=0x05; // start at 5
   }
}
```

Listing 6.3: Example of the switch statement.

For more information on stepper motors, see Section 8.8 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers or Section 6.5 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano..

This next example shows that the multiple tests can be performed for the same condition.

```
// ASCII to decimal digit conversion
unsigned char convert(unsigned char letter){ unsigned char digit;
   switch (letter) {
      case 'A':
      case 'B':
      case 'C':
      case 'D':
      case 'E':
      case 'F':
         digit=letter+10-'A';
         break;
      case 'a':
      case 'b':
      case 'c':
      case 'd':
      case 'e':
      case 'f':
         digit=letter+10-'a';
         break;
      default:
          digit=letter-'0';
   }
   return digit; }
```

Listing 6.4: Example of the switch statement.

The body of the switch is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks. This restriction enforces the C rule that a block containing declarations must be entered through its leading brace.

The While Statement

The while statement is one of three statements that determine the repeated execution of a controlled statement. This statement alone is sufficient for all loop control needs. The other two merely provide an improved syntax and an execute-first feature. While statements have the form

```
while ( ExpressionList ) Statement
```

where ExpressionList is a list of one or more expressions and Statement is an simple or compound statement. If more than one expression is given, the right-most expression yields the value to be tested. First, ExpressionList is evaluated. If it yields true (non-zero), then Statement is executed and ExpressionList is evaluated again. As long as it yields true, Statement executes repeatedly. When it yields false, Statement is skipped, and control continues with whatever follows.

In the example

```
i = 5;
while (i) array[--i] = 0;
```

elements 0 through 4 of array[ ] are set to zero. First i is set to 5. Then as long as it is not zero, the assignment statement is executed. With each execution **i** is decremented before being used as a subscript.

It is common to use the **while** statement to implement busy-wait loops. For information on the UART see Section 8.2 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

```
//------------UART_InChar------------
// Wait for new serial port input
// Input: none
// Output: ASCII code for key typed
unsigned char UART_InChar(void){
  while((UART1_FR_R&0x10) != 0);
  return((unsigned char)(UART1_DR_R&0xFF));
}
//------------UART_OutChar------------
// Output 8-bit to serial port
// Input: letter is an 8-bit ASCII character to be transferred
// Output: none
void UART_OutChar(char data){
  while((UART1_FR_R&0x20) != 0);
  UART1_DR_R = data;
}
```

Listing 6.5: Examples of the while statement.

Continue and break statements are handy for use with the while statement (also helpful for the do and for loops). The continue statement has the form

```
continue;
```

It causes control to jump directly back to the top of the loop for the next evaluation of the controlling expression. If loop controlling statements are nested, then continue affects only the innermost surrounding statement. That is, the innermost loop statement containing the continue is the one that starts its next iteration.

The break statement (described earlier) may also be used to break out of loops. It causes control to pass on to whatever follows the loop controlling statement. If while (or any loop or switch) statements are nested, then break affects only the innermost statement containing the break. That is, it exits only one level of nesting.

The For Statement

The for statement also controls loops. It is really just an embellished while in which the three operations normally performed on loop-control variables (initialize, test, and modify) are brought together syntactically. It has the form

```
for ( ExpressionList? ;
ExpressionList? ;
ExpressionList? ) Statement
```

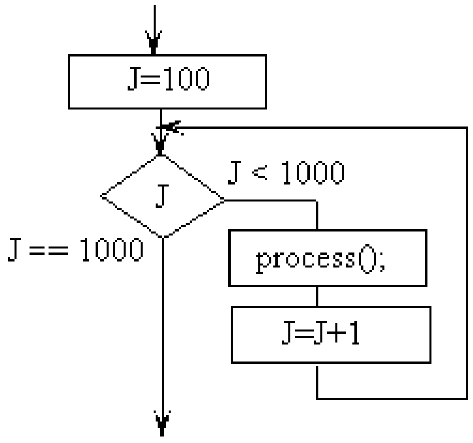
For statements are performed in the following steps:

The first ExpressionList is evaluated. This is done only once to initialize the control variable(s).

The second ExpressionList is evaluated to determine whether or not to perform Statement. If more than one expression is given, the right-most expression yields the value to be tested. If it yields false (zero), control passes on to whatever follows the for statement. But, if it yields true (non-zero), Statement executes.

The third ExpressionList is then evaluated to adjust the control variable(s) for the next pass, and the process goes back to step 2. E.g.,

```
for(J=100;J<1000;J++) { process();}
```



A five-element array is set to zero, could be written as

```
for (i = 4; i >= 0; --i) array[i] = 0;
```

or a little more efficiently as

```
for (i = 5; i; array[--i] = 0) ;
```

Any of the three expression lists may be omitted, but the semicolon separators must be kept. If the test expression is absent, the result is always true. Thus

```
for (;;) {...break;...}
```

will execute until the break is encountered.

As with the while statement, break and continue statements may be used with equivalent effects. A break statement makes control jump directly to whatever follows the for statement. And a continue skips whatever remains in the controlled block so that the third ExpressionList is evaluated, after which the second one is evaluated and tested. In other words, a continue has the same effect as transferring control directly to the end of the block controlled by the for.

The Do Statement

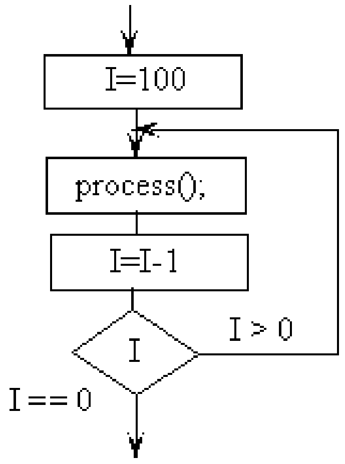
The do statement is the third loop controlling statement in C. It is really just an execute-first while statement. It has the form

```
do Statement while ( ExpressionList ) ;
```

Statement is any simple or compound statement. The do statement executes in the following steps: first Statement is executed. Then, ExpressionList is evaluated and tested. If more than one expression is given, the right most expression yields the value to be tested. If it yields true (non-zero), control goes back to step 1; otherwise, it goes on to whatever follows.

As with the while and for statements, break and continue statements may be used. In this case, a continue causes control to proceed directly down to the while part of the statement for another test of ExpressionList. A break makes control exit to whatever follows the do statement.

```
I=100; do { process(); I--;} while (I>0);
```



The example of the five-element array could be written as

```
i = 4;
do {array[i] = 0; --i;} while (i >= 0);
```

or as

```
i = 4;
do array[i--] = 0; while (i >= 0);
```

or as

```
i = 5;
do array[--i] = 0; while (i);
```

The Return Statement

The return statement is used within a function to return control to the caller. Return statements are not always required since reaching the end of a function always implies a return. But they are required when it becomes necessary to return from interior points within a function or when a useful value is to be returned to the caller. Return statements have the form

```
return ExpressionList? ;
```

ExpressionList? is an optional list of expressions. If present, the last expression determines the value to be returned by the function. If absent, the returned value is unpredictable.

Null Statements

The simplest C statement is the null statement. It has no text, just a semicolon terminator. As its name implies, it does exactly nothing. Why have a statement that serves no purpose? Well, as it turns out, statements that do nothing can serve a purpose. As we saw in Chapter 5, expressions in C can do work beyond that of simply yielding a value. In fact, in C programs, all of the work is accomplished by expressions; this includes assignments and calls to functions that invoke operating system services such as input/output operations. It follows that anything can be done at any point in the syntax that calls for an expression. Take, for example, the statement

```
while((UART1_FR_R&0x20) != 0); /* Wait for TXFF to be set */
```

in which the `((UART1_FR_R&0x20) != 0)` controls the execution of the null statement following. The null statement is just one way in which the C language follows a philosophy of attaching intuitive meanings to seemingly incomplete constructs. The idea is to make the language as general as possible by having the least number of disallowed constructs.

The Goto Statement

Goto statements break the sequential flow of execution by causing control to jump abruptly to designated points. They have the general form **goto Name** where **Name** is the name of a label which must appear in the same function. It must also be unique within the function.

```
short data[10];
void clear(void){ short n;
      n=1;
loop: data[n]=0;
      n++;
      if(n==10) goto done;
      goto loop;
done:
}
```

Listing 6.6: Examples of a goto statements

Notice that labels are terminated with a colon. This highlights the fact that they are not statements but statement prefixes which serve to label points in the logic as targets for goto statements. When control reaches a **goto**, it proceeds directly from there to the designated label. Both forward and backward references are allowed, but the range of the jump is limited to the body of the function containing the **goto** statement.

As we observed above, goto statements, cannot be used in functions which declare locals in blocks which are subordinate to the outermost block of the function.

Because they violate the structured programming criteria, **goto** statements should be used sparingly, if at all. Over reliance on them is a sure sign of sloppy thinking.

Missing Statements

It may be surprising that nothing was said about input/output, program control, or memory management statements. The reason is that such statements do not exist in the C language proper.

In the interest of portability these services have been relegated to a set of standard functions in the run-time library. Since they depend so heavily on the run-time environment, removing them from the language eliminates a major source of compatibility problems. Each implementation of C has its own library of standard functions that perform these operations. Since different compilers have libraries that are pretty much functionally equivalent, programs have very few problems when they are compiled by different compilers.

Go to Chapter 7 on Pointers Return to Table of Contents

Chapter 7: Pointers

What's in Chapter 7?

The ability to work with memory addresses is an important feature of the C language. This feature allows programmers the freedom to perform operations similar to assembly language. Unfortunately, along with the power comes the potential danger of hard-to-find and serious run-time errors. In many situations, array elements can be reached more efficiently through pointers than by subscripting. It also allows pointers and pointer chains to be used in data structures. Without pointers the run-time dynamic memory allocation and deallocation using the heap would not be possible. We will also use a format similar to pointers to develop mechanisms for accessing I/O ports. These added degrees of flexibility are absolutely essential for embedded systems.

Addresses and Pointers

Addresses that can be stored and changed are called *pointers*. A pointer is really just a variable that contains an address. Although, they can be used to reach objects in memory, their greatest advantage lies in their ability to enter into arithmetic (and other) operations, and to be changed. Just like other variables, pointers have a type. In other words, the compiler knows the format (8-bit 16-bit 32-bit, unsigned signed) of the data pointed to by the address.

Not every address is a pointer. For instance, we can write **&var** when we want the address of the variable **var**. The result will be an address that is not a pointer since it does not have a name or a place in memory. It cannot, therefore, have its value altered.

Other examples include an array or a structure name. As we shall see in Chapter 8, an unsubscripted array name yields the address of the array. In Chapter 9, a structure name yields the address of the structure. But, since arrays and structures cannot be moved around in memory, their addresses are not variable. So, although, such addresses have a name, they do not exist as objects in memory (the array does, but its address does not) and cannot, therefore, be changed.

A third example is a character string. Chapter 3 indicated that a character string yields the address of the character array specified by the string. In this case the address has neither a name or a place in memory, so it too is not a pointer.

Pointer Declarations

The syntax for declaring pointers is like that for variables (Chapter 4) except that pointers are distinguished by an asterisk that prefixes their names. Listing 7-1 illustrates several legitimate pointer declarations. Notice, in the third example, that we may mix pointers and variables in a single declaration. I.e., the variable data and the pointer pt3 are declared in the same statement. Also notice that the data type of a pointer declaration specifies the type of object to which the pointer refers, not the type of the pointer itself. As we shall see, all pointers on the Cortex M contain 32-bit unsigned absolute addresses.

```
short *pt1;  /* define pt1, declare as a pointer to a 16-bit integer */
char *pt2;   /* define pt2, declare as a pointer to an 8-bit character */
unsigned short data,*pt3;    /* define data and pt3,
    declare data as an unsigned 16-bit integer and
    declare pt3 as a pointer to a 16-bit unsigned integer */
long *pt4;   /* define pt4, declare as a pointer to a 32-bit integer */
extern short *pt5;    /* declare pt5 as a pointer to an integer */
```

Listing 7-1: Example showing a pointer declarations

The best way to think of the asterisk is to imagine that it stands for the phrase "object at" or "object pointed to by." The first declaration in Listing 7-1 then reads "the object at (pointed to by) **pt1** is a 16-bit signed integer."

Pointer Referencing

We can use the pointer to retrieve data from memory or to store data into memory. Both operations are classified as *pointer references*. The syntax for using pointers is like that for variables except that pointers are distinguished by an asterisk that prefixes their names. Figures 7-1 through 7-4 illustrate several legitimate pointer references. In the first figure, the global variables contain unknown data (actually we know the compiler will zero global variables). The arrow identifies the execution location. Assume addresses 0x20000000 through 0x20000017 exist in RAM.

```
long *pt;       // pointer to 32-bit data
long data;      // 32-bit
long buffer[4]; // array of 4 32-bit numbers
int main(void){
  pt = &buffer[1];
  *pt = 1234;
  data = *pt;
  return 1;
}
address     data        contents
0x20000000  0x00000000  pt
0x20000004  0x00000000  data
0x20000008  0x00000000  buffer[0]
0x2000000C  0x00000000  buffer[1]
0x20000010  0x00000000  buffer[2]
0x20000014  0x00000000  buffer[3]
```
Figure 7-1: Pointer Referencing

The C code `pt=&buffer[1];` will set the `pt` to point to `buffer[1]`. The expression **&buffer[1]** returns the address of the second 32-bit element of the **buffer** (0x2000000C). Therefore the line **pt=&buffer[1];** makes **pt** point to **buffer[1]**.

```
address     data        contents
0x20000000  0x2000000C  pt
0x20000004  0x00000000  data
0x20000008  0x00000000  buffer[0]
0x2000000C  0x00000000  buffer[1]
0x20000010  0x00000000  buffer[2]
0x20000014  0x00000000  buffer[3]
```

The C code `(*pt)=0x1234;` will store **0x1234** into the place pointed to by `pt`. In particular, it stores **0x1234** into `buffer[1]`. When the ***pt** occurs on the left-hand-side of an assignment statement data is stored into memory at the address. Recall the ***pt** means "the 32-bit signed integer at 0x2000000C". I like to add the parentheses () to clarify that *and **pt** are one object. Therefore the line **(*pt)=0x1234;** sets **buffer[1]** to 0x1234.

address	data	contents
0x20000000	0x2000000C	pt
0x20000004	0x00000000	data
0x20000008	0x00000000	buffer[0]
0x2000000C	**0x00001234**	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

The C code `data=(*pt);` will read memory from address pointed to by pointer **pt** into the place pointed to by **`data`**. In particular, it stores **0x1234** into **`data`**. When the ***pt** occurs on the right-hand-side of an assignment statement data is retrieved from memory at the address. Again, I like to add the parentheses () to clarify that * and pt are one object. Therefore the line **data=(*pt);** sets **data** to 0x1234 (more precisely, it copies the 32-bit information from **buffer[1]** into **data**.)

address	data	contents
0x20000000	0x2000000C	pt
0x20000004	**0x00001234**	data
0x20000008	0x00000000	buffer[0]
0x2000000C	0x00001234	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

The following Cortex M assembly was generated by Keil uVision when the above pointer example was compiled.

```
    48:         pt = &buffer[1];
0x000003C4 4806      LDR          r0,[pc,#24]  ; @0x000003E0
0x000003C6 4907      LDR          r1,[pc,#28]  ; @0x000003E4
0x000003C8 6008      STR          r0,[r1,#0x00]
    49:         *pt = 1234;
0x000003CA F24040D2  MOVW         r0,#0x4D2
0x000003CE 6809      LDR          r1,[r1,#0x00]
0x000003D0 6008      STR          r0,[r1,#0x00]
    50:         data = *pt;
0x000003D2 4804      LDR          r0,[pc,#16]  ; @0x000003E4
0x000003D4 6800      LDR          r0,[r0,#0x00]
0x000003D6 6800      LDR          r0,[r0,#0x00]
0x000003D8 4903      LDR          r1,[pc,#12]  ; @0x000003E8
0x000003DA 6008      STR          r0,[r1,#0x00]
    51:         return 1;
0x000003DC 2001      MOVS         r0,#0x01
    52: }
0x000003DE 4770      BX           lr
```

Memory Addressing

The size of a pointer depends on the architecture of the CPU and the implementation of the C compiler. For more information on the architectureof the TM4C see Chapter 3 of Embedded

Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

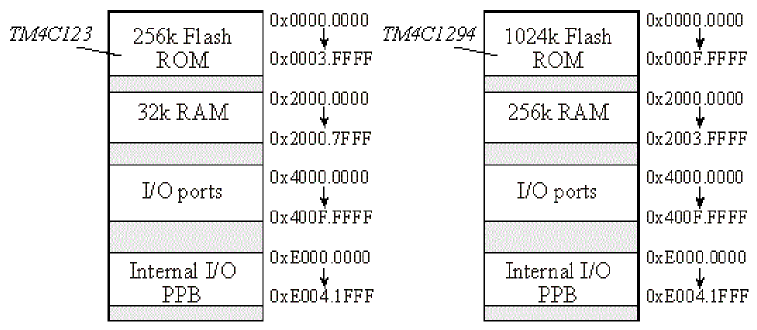


Figure 7-2: Memory map of the TM4C123 and TM4C1294.

Most embedded systems employ a segmented memory architecture. From a physical standpoint we might have a mixture of regular RAM, battery-backed-up RAM, regular EEPROM, flash EPROM, regular PROM, one-time-programmable PROM and ROM. RAM is the only memory structure that allows the program both read and write access. Table 7-1 shows the various types of memory available on most microcomputers. The RAM contains temporary information that is lost when the power is shunt off. This means that all variables allocated in RAM must be explicitly initialized at run time by the software. If the embedded system includes a separate battery for the RAM, then information is not lost when the main power is removed. Some microcomputers have EEPROM. The number of erase/program cycles depends on the memory technology. EEPROM is often used as the main program memory during product development. In the final product we can use EEPROM for configuration constants and even nonvolatile data logging. The one-time-programmable PROM is a simple nonvolatile storage used in small volume products that can be programmed only once with inexpensive equipment. The ROM is a low-cost nonvolatile storage used in large volume products that can be programmed only once at the factory. ***For the number of write cycles available for ROM see the appropriate data sheet for that microcontroller.

Memory	When power is removed	Ability to Read/Write	Program cycles
RAM	volatile	random and fast access	infinite
battery-backed RAM	nonvolatile	random and fast access	infinite
EEPROM	nonvolatile	easily reprogrammed	***
Flash	nonvolatile	easily reprogrammed	***
OTP PROM	nonvolatile	can be easily programmed	once
ROM	nonvolatile	programmed at the factory	once

Table 7-1: Various types of memory available for microntrollers.

In an embedded application, we usually put global variables, the heap, and local variables in RAM because these types of information can change during execution. When software is to be executed on a regular computer, the machine instructions are usually read from a mass storage device (like a disk) and loaded into memory. Because the embedded system usually has no mass storage device, the machine instructions and fixed constants must be stored in nonvolatile memory. If there is both EEPROM and ROM on our microcomputer, we put some fixed constants in EEPROM and some in ROM. If it is information that we may wish to change in the future, we could put it in EEPROM. Examples include language-specific strings, calibration constants, finite state machines, and system ID numbers. This allows us to make minor modifications to the system by reprogramming the EEPROM without throwing the chip away. If our project involves producing a small number of devices then the program can be placed in EPROM or EEPROM. For a project with a large volume it will be cost effective to place the machine instructions in ROM.

Pointer Arithmetic

A major difference between addresses and ordinary variables or constants has to do with the interpretation of addresses. Since an address points to an object of some particular type, adding one (for instance) to an address should direct it to the next object, not necessarily the next byte. If the address points to integers, then it should end up pointing to the next integer. But, since integers occupy two bytes, adding one to an integer address must actually increase the address by two. Likewise, if the address points to long integers, then adding one to an address should end up pointing to the next long integer by increasing the address by four. A similar consideration applies to subtraction. In other words, values added to or subtracted from an address must be scaled according to the size of the objects being addressed. This automatic correction saves the programmer a lot of thought and makes programs less complex since the scaling need not be coded explicitly. The scaling factor for long integers is four; the scaling factor for integers is two; the scaling factor for characters is one. Therefore, character addresses do not receive special handling. It should be obvious that when define structures (see Chapter 9) of other sizes, the appropriate factors would have to be used.

A related consideration arises when we imagine the meaning of the difference of two addresses. Such a result is interpreted as the number of objects between the two addresses. If the objects are integers, the result must be divided by two in order to yield a value which is consistent with this meaning. See Chapter 8 for more on address arithmetic.

When an address is operated on, the result is always another address of the same type. Thus, if **ptr** is a signed 16-bit integer pointer, then **ptr+1** is also points to a signed 16-bit integer.

Precedence determines the order of evaluation. See a table of precedence. One of the most common mistakes results when the programmer meglects the fact the ***** used as a unary pointer reference has precedence over all binary operators. This means the expression ***ptr+1** is the same as **(*ptr)+1** and not ***(ptr+1)**. This is an important point so I'll mention it again, *"When confused about precedence (and aren't we all) add parentheses to clarify the expression."*

Pointer Comparisons

One major difference between pointers and other variables is that pointers are always considered to be unsigned. This should be obvious since memory addresses are not signed. This property of pointers (actually all addresses) ensures that only unsigned operations will be performed on them. It further means that the other operand in a binary operation will also be regarded as unsigned (whether or not it actually is). In the following example, **pt1** and **pt2**[5] return the current values of the addresses. For instance, if the array **pt2**[] contains addresses, then it would make sense to write

```
short *pt1;      /* define 16-bit integer pointer */
short *pt2[10];  /* define ten 16-bit integer pointers */
short done(void){  /* returns true if pt1 is higher than pt2[5] */
    if(pt1>pt2[5]) return(1);
    return(0);
}
```

Listing 7-2: Example showing a pointer comparisons

which performs an unsigned comparison since pt1 and pt2 are pointers. Thus, if pt2[5] contains 0x2000F000 and pt1 contains 0x20001000, the expression will yield true, since 0x2000F000 is a higher unsigned value than 0x20001000.

It makes no sense to compare a pointer to anything but another address or zero. C guarantees that valid addresses can never be zero, so that particular value is useful in representing the absence of an address in a pointer.

Furthermore, to avoid portability problems, only addresses within a single array should be compared for relative value (e.g., which pointer is larger). To do otherwise would necessarily involve assumptions about how the compiler organizes memory. Comparisons for equality, however, need not observe this restriction, since they make no assumption about the relative positions of objects. For example if **pt1** points into one data array and **pt2** points into a different array, then comparing **pt1** to **pt2** would be meaningless. Which pointer is larger would depend on where in memory the two arrays were assigned.

A FIFO Queue Example

To illustrate the use of pointers we will design a two-pointer FIFO. The first in first out circular queue (FIFO) is also useful for data flow problems. It is a very common data structure used for I/O interfacing. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFO's studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the source and sink processes. Without the FIFO we would have to produce 1 piece of data, then process it, produce another piece of data, then process it. With the FIFO, the source process can continue to produce data without having to wait for the sink to finish processing the previous data. This decoupling can significantly improve system performance.

GETPT points to the data that will be removed by the next call to GET, and PUTPT points to the empty space where the data will stored by the next call to PUT. If the FIFO is full when PUT is called then the subroutine should return a full error (e.g., V=1.) Similarly, if the FIFO is empty when GET is called, then the subroutine should return an empty error (e.g., V=1.) The PUTPT and GETPT must be wrapped back up to the top when they reach the bottom.

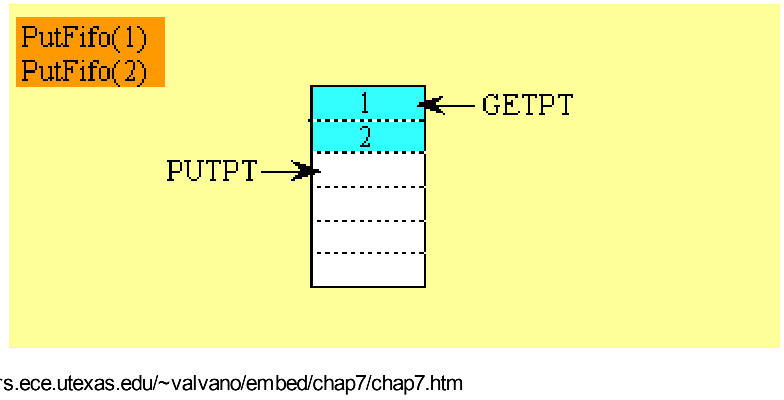


Figure 7-3: Fifo example showing the PUTPT and GETPT wrap.

There are two mechanisms to determine whether the FIFO is empty or full. A simple method is to implement a counter containing the number of bytes currently stored in the FIFO. GET would decrement the counter and PUT would increment the counter. The second method is to prevent the FIFO from being completely full. For example, if the FIFO had 100 bytes allocated, then the PUT subroutine would allow a maximum of 99 bytes to be stored. If there were already 99 bytes in the FIFO and another PUT were called, then the FIFO would not be modified and a full error would be returned. In this way if PUTPT equals GETPT at the beginning of GET, then the FIFO is empty. Similarly, if PUTPT+1 equals GETPT at the beginning of PUT, then the FIFO is full. Be careful to wrap the PUTPT+1 before comparing it to GETPT. This second method does not require the length to be stored or calculated.

```
/* Pointer implementation of the FIFO */
#define FifoSize 10 /* Number of 8 bit data in the Fifo */
char *PUTPT;    /* Pointer of where to put next */
char *GETPT;    /* Pointer of where to get next */
/* FIFO is empty if PUTPT=GETPT */
/* FIFO is full if PUTPT+1=GETPT */
char Fifo[FifoSize]; /* The statically allocated fifo data */
void InitFifo(void) {
    PUTPT=GETPT=&Fifo[0]; /* Empty when PUTPT=GETPT */
}
int PutFifo (char data) { char *Ppt; /* Temporary put pointer */
    Ppt=PUTPT; /* Copy of put pointer */
    *(Ppt++)=data; /* Try to put data into fifo */
    if (Ppt == &Fifo[FifoSize]) Ppt = &Fifo[0]; /* Wrap */
    if (Ppt == GETPT ){
        return(0);}   /* Failed, fifo was full */
    else{
        PUTPT=Ppt;
        return(-1);   /* Successful */
    }
}
int GetFifo (char *datapt) {
    if (PUTPT== GETPT){
        return(0);}   /* Empty if PUTPT=GETPT */
    else{
        *datapt=*(GETPT++);
        if (GETPT == &Fifo[FifoSize])
            GETPT = &Fifo[0];
        return(-1);
    }
}
```

Listing 7-3: Fifo queue implemented with pointers

Since these routines have read modify write accesses to global variables the three functions (InitFifo, PutFifo, GetFifo) are themselves not reentrant. Consequently interrupts are temporarily disabled, to prevent one thread from reentering these Fifo functions. One advantage of this pointer implementation is that if you have a single thread that calls the GetFifo (e.g., the main program) and a single thread that calls the PutFifo (e.g., the serial port receive interrupt handler), then this PutFifo function can interrupt this GetFifo function without loss of data. So in this particular situation, interrupts would not have to be disabled. It would also operate properly if there were a single interrupt thread calling GetFifo (e.g., the serial port transmit interrupt handler) and a single thread calling PutFifo (e.g., the main program.) On the other hand, if the situation is more general, and multiple threads could call PutFifo or multiple threads could call GetFifo, then the interrupts would have to be temporarily disabled.

I/O Port Access

Even though the mechanism to access I/O ports technically does not fit the definition of pointer, it is included in this chapter because it involves addresses. The line `NVIC_ST_RELOAD_R = delay-1;` generates a 32-bit I/O write operation to the port at address 0xE000E014. The `GPIO_PORTF_DATA_R` on the right side of an assignment statement generates a 32-bit read from address 0x400253FC. The `NVIC_ST_CTRL_R` in the while loop generates a 32-bit I/O read operation from the port at address 0xE000E010.

```
#define NVIC_ST_CTRL_R          (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R        (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R       (*((volatile unsigned long *)0xE000E018))
#define GPIO_PORTF_DATA_R       (*((volatile unsigned long *)0x400253FC))
// The delay parameter is in units of the 80 MHz core clock.
(12.5 ns)
void SysTick_Wait(unsigned long delay){ unsigned long data;
  NVIC_ST_RELOAD_R = delay-1;  // number of counts to wait
  NVIC_ST_CURRENT_R = 0;       // any value written to
CURRENT clears
  data = GPIO_PORTF_DATA_R;
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count
flag
  }
}
```

Listing 7-5: Sample Program that accesses I/O ports

To understand the port definitions in C, we remember **`#define`** is simply a copy paste. E.g.,

```
#define PA5   (*((volatile unsigned long *)0x40004080))
data = PA5;
```

becomes

```
data = (*((volatile unsigned long *)0x40004080));
```

To understand why we define ports this way, let's break this port definition into pieces. First, 0x40004080 is the address of Port A bit 5. If we write just **`#define PA5 0x40004080`** it will create

```
data = 0x40004080;
```

which does not read the contents of PA5 as desired. This means we need to dereference the address. If we write **`#define PA5 (*0x40004080)`** it will create

```
data = (*0x40004080);
```

This will attempt to read the contents at 0x40004080, but doesn't know whether to read 8, 16, or 32 bits. So the compiler gives a syntax error because the type of data does not match the type of (*0x40004080). To solve a type mismatch in C we **typecast**, placing a (new type) in front of the object we wish to convert. We wish force the type conversion to unsigned 32 bits, so we modify the definition to include the typecast.

The **`volatile`** is added because the value of a port can change beyond the direct action of the software. It forces the C compiler to read a new value each time through a loop and not rely on the previous value.

```
#define PA5   (*((volatile unsigned long *)0x40004080))
void wait(void){
  while((PA5&0x20)==0){};
}
void wait2(void){
  while(((*((volatile unsigned long *)0x40004080))&0x20)==0)
{};
}
void wait3(void){
  volatile unsigned long *pt;
  pt = ((volatile unsigned long *)0x40004080);
  while(((*pt)&0x20)==0){};
}
```

Listing 7-6: Program that accesses I/O ports using pointers

The function `wait3` first sets the I/O pointer then accesses the I/O port indirectly through the pointer.

Go to Chapter 8 on Arrays and Strings Return to Table of Contents

Chapter 8: Arrays and Strings

What's in Chapter 8?

An array is a collection of like variables that share a single name. The individual elements of an array are referenced by appending a *subscript*, in square brackets, behind the name. The subscript itself can be any legitimate C expression that yields an integer value, even a general expression. Therefore, arrays in C may be regarded as collections of like variables. Although arrays represent one of the simplest data structures, it has wide-spread usage in embedded systems.

Strings are similar to arrays with just a few differences. Usually, the array size is fixed, while strings can have a variable number of elements. Arrays can contain any data type (**char short int** even other arrays) while strings are usually ASCII characters terminated with a NULL (0) character. In general we allow random access to individual array elements. On the other hand, we usually process strings sequentially character by character from start to end. Since these differences are a matter of semantics rather than specific limitations imposed by the syntax of the C programming language, the descriptions in this chapter apply equally to data arrays and character strings. String literals were discussed earlier in Chapter 3; in this chapter we will define data structures to hold our strings. In addition, C has a rich set of predefined functions to manipulate strings.

Array Subscripts

When an array element is referenced, the subscript expression designates the desired element by its position in the **data**. The first element occupies position zero, the second position one, and so on. It follows that the last element is subscripted by [N-1] where N is the number of elements in the array. The statement

```
data[9] = 0;
```

for instance, sets the tenth element of **data** to zero. The array subscript can be any expression that results in a 16-bit integer. The following for-loop clears 100 elements of the array **data** to zero.

```
for(j=0; j<100; j++) data[j] = 0;
```

If the array has two dimensions, then two subscripts are specified when referencing. As programmers we may any assign logical meaning to the first and second subscripts. For example we could consider the first subscript as the row and the second as the column. Then, the statement

```
ThePosition = position[3][5];
```

copies the information from the 4th row 6th column into the variable **ThePosition**. If the array has three dimensions, then three subscripts are specified when referencing. Again we may any assign logical meaning to the various subscripts. For example we could consider the first subscript as the x coordinate, the second subscript as the y coordinate and the third subscript as the z coordinate. Then, the statement

```
humidity[2][3][4] = 100;
```

sets the humidity at point (2,3,4) to 100. Array subscripts are treated as signed 32-bit integers. It is the programmer's responsibility to see that only positive values are produced, since a negative subscript would refer to some point in memory preceding the array. One must be particularly careful about assuming what existing either in front of or behind our arrays in memory.

Array Declarations

Just like any variable, arrays must be declared before they can be accessed. The number of elements in an array is determined by its declaration. Appending a constant expression in square brackets to a name in a declaration identifies the name as the name of an array with the number of elements indicated. Multi-dimensional arrays require multiple sets of brackets. The examples in Listing 8-1 are valid declarations.

```
short data[5];         /* define data, allocate space for 5 16-bit integers */
char string[20];       /* define string, allocate space for 20 8-bit characters */
int time,width[6];     /* define time, width, allocate space for 16-bit characters
*/
short xx[10][5];       /* define xx, allocate space for 50 16-bit integers */
short pts[5][5][5];    /* define pts, allocate space for 125 16-bit integers */
extern char buffer[]; /* declare buffer as an external character array */
```

Listing 8-1: Example showing a array declarations

Notice in the third example that ordinary variables may be declared together with arrays in the same statement. In fact array declarations obey the syntax rules of ordinary declarations, as described in Chapters 4 and 7, except that certain names are designated as arrays by the presence of a dimension expression.

Notice the size of the external array, **buffer[]**, is not given. This leads to an important point about how C deals with array subscripts. The array dimensions are only used to determine how much memory to reserve. **It is the programmer's responsibility to stay within the proper bounds.** In particular, you must not let the subscript become negative for above N-1, where N is the size of the array.

Another situation in which an array's size need not be specified is when the array elements are given initial values. As we will see in Chapter 9, the compiler will determine the size of such an array from the number of initial values.

Array References

In C we may refer to an array in several ways. Most obviously, we can write subscripted references to array elements, as we have already seen. C interprets an unsubscripted array name as the address of the array. In the following example, the first two lines set **x**to equal the value of the first element of the array. The third and fourth lines both set **pt** equal to the address of the array. Chapter 7 introduced the address operator & that yields the address of an object. This operator may also be used with array elements. Thus, the expression **&data[3]** yields the address of the fourth element. Notice too that **&data[0]** and **data+0** and **data** are all equivalent. It should be clear by analogy that &**data**[3] and **data**+3 are also equivalent.

```
short x,*pt,data[5]; /* a variable, a pointer, and an array */
void Set(void){
  x = data[0];    /* set x equal to the first element of data */
  x = *data;      /* set x equal to the first element of data */
  pt = data;      /* set pt to the address of data */
  pt = &data[0];  /* set pt to the address of data */
  x = data[3];    /* set x equal to the fourth element of data */
  x = *(data+3);  /* set x equal to the fourth element of data */
  pt = data+3;    /* set pt to the address of the fourth element */
  pt = &data[3];  /* set pt to the address of the fourth element */
}
```

Listing 8-2: Example showing array references

Pointers and Array Names

The examples in the section suggest that pointers and array names might be used interchangeably. And, in many cases, they may. C will let us subscript pointers and also use array names as addresses. In the following example, the pointer **pt** contains the address of an array of integers. Notice the expression **pt[3]** is equivalent to ***(pt+3)**.

```
short *pt,data[5]; /* a pointer, and an array */
void Set(void){
  pt = data;      /* set pt to the address of data */
  data[2] = 5;    /* set the third element of data to 5 */
  pt[2] = 5;      /* set the third element of data to 5 */
  *(pt+2) = 5;    /* set the third element of data to 5 */
}
```

Listing 8-3: Example showing pointers to access array elements

It is important to realize that although C accepts unsubscripted array names at addresses, they are not the same as pointers. In the following example, we can not place the unsubscripted array name on the left-hand-side of an assignment statement.

```
short buffer[5],data[5]; /* two arrays */
void Set(void){
  data = buffer;      /* illegal assignment */
}
```

Listing 8-4: Example showing an illegal array assignment

Since the unsubscripted array name is its address, the statement **data=buffer;** is an attempt to change its address. What sense would that make? The array, like any object, has a fixed home in memory; therefore, its address cannot be changed. We say that array is not a ***lvalue***; that is, it cannot be used on the left side of an assignment operator (nor may it be operated on by increment or decrement operators). It simply cannot be changed. Not only does this assignment make no sense, it is physically impossible because an array address is not a variable. There is no place reserved in memory for an array's address to reside, only the elements.

Negative Subscripts

Since a pointer may point to any element of an array, not just the first one, it follows that negative subscripts applied to pointers might well yield array references that are in bounds. This sort of thing might be useful in situations where there is a relationship between successive elements in an array and it becomes necessary to reference an element preceding the one being pointed to. In the following example, **data** is an array containing time-dependent (or space-dependent) information. If pt points to an element in the array, **pt[-1]** is the previous element and **pt[1]** is the following one. The function calculates the second derivative using a simple discrete derivative.

```
short *pt,data[100]; /* a pointer and an array */
void CalcSecond(void){ short d2Vdt2;
  for(pt=data+1; pt<data+99; pt++){
    d2Vdt2 = (pt[-1]-2*pt[0]+pt[1]);
  }
}
```

Listing 8-5: Example showing negative array subscripting

Address Arithmetic

As we have seen, addresses (pointers, array names, and values produced by the address operator) may be used freely in expressions. This one fact is responsible for much of the power of C.

As with pointers (Chapter 7), all addresses are treated as unsigned quantities. Therefore, only unsigned operations are performed on them. Of all the arithmetic operations that could be performed on addresses only two make sense, displacing an address by a positive or negative amount, and taking the difference between two addresses. All others, though permissible, yield meaningless results.

Displacing an address can be done either by means of subscripts or by use of the plus and minus operators, as we saw earlier. These operations should be used only when the original address and the displaced address refer to positions in the same array or data structure. Any other situation would assume a knowledge of how memory is organized and would, therefore, be ill-advised for portability reasons.

As we saw in Chapter 7, taking the difference of two addresses is a special case in which the compiler interprets the result as the number of objects lying between the addresses.

String Functions in string.h

Most compliers implement many useful string manipulation functions. Recall that strings are 8-bit arrays with a null-termination. The prototypes for these functions can be found in the **string.h** file. You simply include this file whenever you wish to use any of these routines. The rest of this section explains the functions one by one. Most compilers for the ARM Cortex M treat each of the counts as an unsigned 32-bit integer.

```
typedef unsigned int size_t;
void *memchr(void *, int, size_t);
int memcmp(void *, void *, size_t);
```

```
void *memcpy(void *, void *, size_t);
void *memmove(void *, void *, size_t);
void *memset(void *, int, size_t);
char *strcat(char *, const char *);
char *strchr(const char *, int);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
char *strcpy(char *, const char *);
size_t strcspn(const char *, const char *);
size_t strlen(const char *);
char *strncat(char *, const char *, size_t);
int strncmp(const char *, const char *, size_t);
char *strncpy(char *, const char *, size_t);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
size_t strspn(const char *, const char *);
char *strstr(const char *, const char *);
```

Listing 8-6: Prototypes for string functions

The first five functions are general-purpose memory handling routines.

```
void *memchr(void *p, int c, size_t n);
```

Starting in memory at address p, **memchr** will search for the first unsigned 8-bit byte that matches the value in **c**. At most n bytes are searched. If successful, a pointer to the 8-bit byte is returned, otherwise a NULL pointer is returned.

```
int memcmp(void *p, void *q, size_t n);
```

Assuming the two pointers are directed at 8-bit data blocks of size **n**, **memcmp** will return a negative value if the block pointed to by **p**is lexicographically less than the block pointed to by **q**. The return value will be zero if they match, and positive if the block pointed to by **p**is lexicographically greater than the block pointed to by **q**.

```
void *memcpy(void *dst, void *src, size_t n);
```

Assuming the two pointers are directed at 8-bit data blocks of size n, **memcpy** will copy the data pointed to by pointer **src**, placing it in the memory block pointed to by pointer **dst**. The pointer **dst** is returned.

```
void *memmove(void *dst, void *src, size_t);
```

Assuming the two pointers are directed at 8-bit data blocks of size n, **memmove** will copy the data pointed to by pointer **src**, placing it in the memory block pointed to by pointer **dst**. This routine works even if the blocks overlap. The pointer **dst** is returned.

```
void *memset(void *p, int c, size_t n);
```

Starting in memory at address p, **memset** will set n 8-bit bytes to the 8-bit value in c. The pointer **p**is returned.

The remaining functions are string-handling routines.

```
char *strcat(char *p, const char *q);
```

Assuming the two pointers are directed at two null-terminated strings, **strcat** will append a copy of the string pointed to by pointer **q**, placing it the end of the string pointed to by pointer **p**. The pointer **p**is returned. It is the programmer's responsibility to ensure the destination buffer is large enough.

```
char *strchr(const char *p, int c);
```

Assuming the pointer is directed at a null-terminated string. Starting in memory at address p, **strchr** will search for the first unsigned 8-bit byte that matches the value in c. It will search until a match is found or stop at the end of the string. If successful, a pointer to the 8-bit byte is returned, otherwise a NULL pointer is returned.

```
int strcmp(const char *p, const char *q);
```

Assuming the two pointers are directed at two null-terminated strings, **strcmp** will return a negative value if the string pointed to by **p**is lexicographically less than the string pointed to by **q**. The return value will be zero if they match, and positive if the string pointed to by **p**is lexicographically greater than the string pointed to by **q**.

```
char *strcpy(char *dst, const char *src);
```

We assume **scr** points to a null-terminated string and **dst** points to a memory buffer large enough to hold the string. **strcpy** will copy the string (including the null) pointed to by **src**, into the buffer pointed to by pointer **dst**. The pointer **dst** is returned. It is the programmer's responsibility to ensure the destination buffer is large enough.

```
size_t strcspn(const char *p, const char *q);
```

The string function **strcspn** will compute the length of the maximal initial substring within the string pointed to by **p**that has no characters in common with the string pointed to by **q**. For example the following call returns the value 5.

```
n=strcspn("label: ldaa 10,x ;comment"," ;:*\n\t\l");
```

A common application of this routine is parsing for tokens. The first parameter is a line of text and the second parameter is a list of delimiters (e.g., space, semicolon, colon, star, return, tab and linefeed). The function returns the length of the first token (i.e., the size of *label*).

```
size_t strlen(const char *p);
```

The string function **strlen** returns the length of the string pointed to by pointer **p**. The length is the number of characters in the string not counting the null-termination.

```
char *strncat(char *p, const char *q, size_t n);
```

This function is similar to **strcat**. Assuming the two pointers are directed at two null-terminated strings, **strncat** will append a copy of the string pointed to by pointer **q**, placing it the end of the string pointed to by pointer **p**. The parameter **n**limits the number of characters, not including the null that will be copied. The pointer **p**is returned. It is the programmer's responsibility to ensure the destination buffer is large enough.

```
int strncmp(const char *p, const char *q, size_t n);
```

This function is similar to **strcmp**. Assuming the two pointers are directed at two null-terminated strings, **strncmp** will return a negative value if the string pointed to by **p** is lexicographically less than the string pointed to by **q**. The return value will be zero if they match, and positive if the string pointed to by **p**is lexicographically greater than the string pointed to by **q**. The parameter **n**limits the number of characters, not including the null that will be compared. For example, the following function call will return a zero because the first 6 characters are the same:

```
n=strncmp("TM4C123,"TM4C1294",6);
```

The following function is similar to **strcpy**.

```
char *strncpy(char *dst, const char *src, size_t n);
```

We assume **scr** points to a null-terminated string and **dst** points to a memory buffer large enough to hold the string. **strncpy** will copy the string (including the null) pointed to by **src**, into the buffer pointed to by pointer **dst**. The pointer **dst** is returned. The parameter **n**limits the number of characters, not including the null that will be copied. If the size of the string pointed to by **src** is equal to or larger than **n**, then the null will not be copied into the buffer pointer to by **dst**. It is the programmer's responsibility to ensure the destination buffer is large enough.

```
char *strpbrk(const char *p, const char *q);
```

This function, **strpbrk** , is called *pointer to break*. The function will search the string pointed to by **p** for the first instance of any of the characters in the string pointed to by **q**. A pointer to the found character is returned. If the search fails to find any characters of the string pointed to by **q** in the string pointed to by **p**, then a null pointer is returned. For example the following call returns a pointer to the colon.

```
pt=strpbrk("label: LDR R0,[R1] ;comment"," ;:*\n\t\l");
```

This function, like **strcspn**, can be used for parsing tokens.

```
char *strrchr(const char *p, int c);
```

The function **strrchr** will search the string pointed to by **p** from the right for the first instance of the character in **c**. From the right means search from back of the string towards the front. A pointer to the found character is returned. If the search fails to find any characters with the 8-bit value **c** in the string pointed to by **p**, then a null pointer is returned. For example the following calls set the **pt1** to point to the 'R' in *LDR* and **pt2** to point to the 'R' in *R1*

```
pt1=strchr("label: LDR R0,[R1] ;comment",'R');
pt2=strrchr("label: LDR R0,[R1] ;comment",'R');
```

Notice that **strchr** searches from the left while **strrchr** searches from the right .

```
size_t strspn(const char *p, const char *q);
```

The function **strspn** will return the length of the maximal initial substring in the string pointed to by **p**that consists entirely of characters in the string pointed to by **q**. In the following example the second string contains the valid set of hexadecimal digits. The function call will return 6 because there is a valid 6-digit hexadecimal string at the start of the line.

```
n=strspn("A12F05+12BAD*45","01234567890ABCDEF");
```

```
char *strstr(const char *p, const char *q);
```

The function **strstr** will search the string pointed to by **p**from the left for the first instance of the string pointed to by **q**. A pointer to the found substring within the first string is returned. If the search fails to find a match, then a null pointer is returned. For example the following calls set the **pt** to point to the 'L' in *LDR*.

```
pt=strstr("label: LDR R0,[R1] ;comment","LDR");
```

A FIFO Queue Example using indices

Another method to implement a statically allocated first-in-first-out FIFO is to use indices instead of pointers. This method is necessary for compilers that do not support pointers. The purpose of this example is to illustrate the use of arrays and indices. Just like the previous FIFO, this is used for order-preserving temporary storage. The function **Fifo_Put** will enter one 8-bit byte into the queue, and **Fifo_Get** will remove one byte. If you call **Fifo_Put** while the FIFO is full (**Size** is equal to **FifoSize**), the routine will return a zero. Otherwise, **Fifo_Put** will save the data in the queue and return a one. The index **PutI** specifies where to put the next 8-bit data. The routine **Fifo_Get** actually returns two parameters. The queue status is the regular function return parameter, while the data removed from the queue is return by reference. I.e., the calling routine passes in a pointer, and **Fifo_Get** stores the removed data at that address. If you call **Fifo_Get** while the FIFO is empty (**Size** is equal to zero), the routine will return a zero. Otherwise, **Fifo_Get** will return the oldest data from the queue and return a one. The index **GetI** specifies where to get the next 8-bit data. The following FIFO implementation uses two indices and a counter. Because of the read-modify-write to `Size`, these routines have a critical section. For information on critical sections see Section 5.3 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

```
/* Index,counter implementation of the FIFO */
#define FifoSize 10    /* Number of 8 bit data in the Fifo */
unsigned char PutI;    /* Index of where to put next */
unsigned char GetI;    /* Index of where to get next */
unsigned char Size;    /* Number currently in the FIFO */
               /* FIFO is empty if Size=0 */
               /* FIFO is full  if Size=FifoSize */
char Fifo[FifoSize];   /* The statically allocated data */
void Fifo_Init(void) {
  PutI=GetI=Size=0;    /* Empty when Size==0 */
}
int Fifo_Put (char data) {
  if (Size == FifoSize )
    return(0);     /* Failed, fifo was full */
  else{
    Size++;
    Fifo[PutI++]=data;      /*  put data into fifo */
    if (PutI == FifoSize) PutI = 0;  /* Wrap */
    return(-1);      /* Successful */
  }
}
int Fifo_Get (char *datapt) {
  if (Size == 0 )
    return(0);     /* Empty if Size=0 */
  else{
    *datapt=Fifo[GetI++];
    Size--;
    if (GetI == FifoSize) GetI = 0;
    return(-1); }
}
```

Listing 8-7: FIFO implemented with two indices and a counter

For information on FIFO queues see either Section 11.3 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano, or Section 3.7 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

Go to Chapter 9 on Structures Return to Table of Contents

Chapter 9: Structures

What's in Chapter 9?

A structure is a collection of variables that share a single name. In an array, each element has the same format. With structures we specify the types and names of each of the elements or members of the structure. The individual members of a structure are referenced by their subname. Therefore, to access data stored in a structure, we must give both the name of the collection and the name of the element. Structures are one of the most powerful features of the C language. In the same way that functions allow us to extend the C language to include new operations, structures provide a mechanism for extending the data types. With structures we can add new data types derived from an aggregate of existing types.

Structure Declarations

Like other elements of C programming, the structure must be declared before it can be used. The declaration specifies the tagname of the structure and the names and types of the individual members. The following example has three members: one 8-bit integer and two word pointers

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // pointer to its address
   unsigned long volatile *ddr;}; // pointer to its direction reg
```

The above declaration does not create any variables or allocate any space. Therefore to use a structure we must define a global or local variable of this type. The tagname (**theport**) along with the keyword **struct** can be used to define variables of this new data type:

```
struct theport PortA,PortB,PortE;
```

The above line defines the three variables and allocates 9 bytes for each of variable. Because the pointers will be 32-bit aligned the compiler will skip three bytes between mask and addr, so each object will occupy 12 bytes. If you knew you needed just three copies of structures of this type, you could have defined them as

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;
   unsigned long volatile *ddr;}PortA,PortB,PortE;
```

Definitions like the above are hard to extend, so to improve code reuse we can use **typedef** to actually create a new data type (called **port** in the example below) that behaves syntactically like **char int short** etc.

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // address
   unsigned long volatile *ddr;}; // direction reg
typedef struct theport port_t;
```

```
port_t PortA,PortB,PortE;
```

Once we have used **typedef** to create **port**_t, we don't need access to the name **theport** anymore. Consequently, some programmers use to following short-cut:

```
typedef struct {
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;        // address
   unsigned long volatile *ddr;}port_t; // direction reg
port_t PortA,PortB,PortE;
```

Similarly, I have also seen the following approach to creating structures that uses the same structure name as the **typedef** name:

```
struct port{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // address
   unsigned long volatile *ddr;}; // direction reg
typedef struct port port;
port PortA,PortB,PortE;
```

Most compilers support all of the above methods of declaring and defining structures.

Accessing Members of a Structure

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure. The following examples show accesses to individual members:

```
PortB.mask = 0xFF;     // the TM4C123 has 8 bits on PORTB
PortB.addr = (unsigned long volatile *)(0x400053FC);
PortB.ddr = (unsigned long volatile *)(0x40005400);
PortE.mask = 0x3F;      // the TM4C123 has 6 bits on PORTE
PortE.addr = (unsigned long volatile *)(0x400243FC);
PortE.ddr = (unsigned long volatile *)(0x40024400);
(*PortE.ddr) = 0;               // specify PortE as inputs
(*PortB.addr) = (*PortE.addr);  // copy from PortE to PortB
```

The syntax can get a little complicated when a member of a structure is another structure as illustrated in the next example:

```
struct theline{
   int x1,y1;    // starting point
   int x2,y2;    // starting point
   unsigned char color;}; // color
typedef struct theline line_t;
struct thepath{
   line_t L1,L2;  // two lines
   char direction;};
typedef struct thepath path_t;
path_t p;        // global
void Setp(void){ line_t myLine; path_t q;
   p.L1.x1 = 5;  // black line from 5,6 to 10,12
   p.L1.y1 = 6;
   p.L1.x2 = 10;
   p.L1.y2 = 12;
   p.L1.color = 255;
   p.L2.x1 = 0;  // white line from 0,1 to 2,3
   p.L2.y1 = 1;
   p.L2.x2 = 2;
```

```
    p.L2.y2 = 3;
    p.L2.color = 0;
    p.direction = -1;
    myLine = p.L1;
    q = p;
};
```

Listing 9-1: Examples of accessing structures

The local variable declaration **line myLine;** will allocate 9 bytes on the stack while **path q;** will allocate 19 bytes on the stack. In actuality most C compilers in an attempt to maintain addresses as 32-bit aligned numbers will actually allocate 12 and 28 bytes respectively. In particular, the Cortex M executes faster if accesses occur on word-aligned addresses. For example, a 32-bit data access to an odd address requires two bus cycles, while a 32-bit data access to a word-aligned address requires only one bus cycle. Notice that the expression **p.L1.x1** is of the type **int**, the term **p.L1** has the type **line**, while just **p** has the type **path**. The expression **q=p;** will copy the entire 15 bytes that constitute the structure from **p** to **q**.

Initialization of a Structure

Just like any variable, we can specify the initial value of a structure at the time of its definition.

```
path_t thePath={{0,0,5,6,128},{5,6,-10,6,128},1};
line_t theLine={0,0,5,6,128};
port_t PortE={0x3F,
   (unsigned long volatile *)(0x400243FC),
   (unsigned long volatile *)(0x40024400)};
```

If we leave part of the initialization blank it is filled with zeros.

```
path_t thePath={{0,0,5,6,128},};
line_t thePath={5,6,10,12,};
port_t PortE={1,  (unsigned char volatile *)(0x100A),};
```

To place a structure in ROM, we define it as a global constant. In the following example the structure fsm[3] will be allocated and initialized in ROM-space. The linked-structure finite syatem machine is a good example of a ROM-based structure. For more information about finite state machines see either Section 6.5 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano, or Section 3.5 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

```
struct State{
    unsigned char Out;     /* Output to Port B */
    unsigned short Wait;   /* Time (62.5ns cycles) to wait */
    unsigned char AndMask[4];
    unsigned char EquMask[4];
    const struct State *Next[4];};  /* Next states */
typedef const struct State state_t;
typedef state_t * StatePtr;
#define stop &fsm[0]
#define turn &fsm[1]
#define bend &fsm[2]
state_t fsm[3]={
{0x34, 16000,   // stop 1 ms
```

```
      {0xFF,   0xF0,   0x27,   0x00},
      {0x51,   0xA0,   0x07,   0x00},
      {turn,   stop,   turn,   bend}},
   {0xB3,40000,   // turn 2.5 ms
      {0x80,   0xF0,   0x00,   0x00},
      {0x00,   0x90,   0x00,   0x00},
      {bend,   stop,   turn,   turn}},
   {0x75,32000,   // bend 2 ms
      {0xFF,   0x0F,   0x01,   0x00},
      {0x12,   0x05,   0x00,   0x00},
      {stop,   stop,   turn,   stop}}};
```

Listing 9-2: Example of initializing a structure in ROM

Using pointers to access structures

Just like other variables we can use pointers to access information stored in a structure. The syntax is illustrated in the following examples:

```
void Setp(void){ path_t *ppt;
   ppt = &p;        // pointer to an existing global variable
   ppt->L1.x1 = 5;  // black line from 5,6 to 10,12
   ppt->L1.y1 = 6;
   ppt->L1.x2 = 10;
   ppt->L1.y2 = 12;
   ppt->L1.color = 255;
   ppt->L2.x1 = 0;  // white line from 0,1 to 2,3
   ppt->L2.y1 = 1;
   ppt->L2.x2 = 2;
   ppt->L2.y2 = 3;
   ppt->L2.color = 0;
   ppt->direction = -1;
   (*ppt).direction = -1;
};
```

Listing 9-3: Examples of accessing a structure using a pointer

Notice that the syntax **ppt->direction** is equivalent to **(*ppt).direction**. The parentheses in this access are required, because along with () and [], the operators **.** and **->** have the highest precedence and associate from left to right. Therefore ***ppt.direction** would be a syntax error because **ppt.direction** can not be evaluated.

As an another example of pointer access consider the finite state machine controller for the fsm[3] structure shown above. The state machine is illustrated in Figure 9-1, and the program shown in Listing 9-4. There is an example in Chapter 10 that extends this machine to implement function pointers.

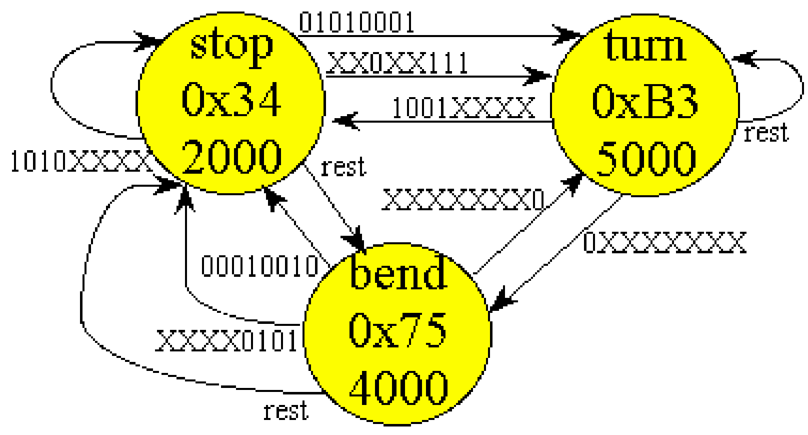


Figure 9-1: State machine

```
void control(void){ StatePtr Pt;
 unsigned char Input; unsigned int i;
  SysTick_Init();
  Port_Init();
  Pt = stop;       // Initial State
  while(1){
    GPIO_PORTA_DATA_R = Pt->Out; // 1) output
    SysTick_Wait(Pt->Wait);      // 2) wait
    Input = GPIO_PORTB_DATA_R;   // 3) input
    for(i=0;i<4;i++)
      if((Input&Pt->AndMask[i])==Pt->EquMask[i]){
        Pt = Pt->Next[i]; // 4) next depends on input
        i=4; }}};
```

Listing 9-4: Finite state machine controller for TM4C123

Passing Structures to Functions

Like any other data type, we can pass structures as parameters to functions. Because most structures occupy a large number of bytes, it makes more sense to pass the structure by reference rather than by value. In the following "call by value" example, the entire 12-byte structure is copied on the stack when the function is called.

```
unsigned long Input(port_t thePort){
   return (*thePort.addr);}
```

When we use "call by reference", a pointer to the structure is passed when the function is called.

```
typedef const struct {
  unsigned char mask;    // defines which bits are active
  unsigned long volatile *addr;      // address
  unsigned long volatile *ddr;}port; // direction reg
port_t PortE={0x3F,
   (unsigned long volatile *)(0x400243FC),
   (unsigned long volatile *)(0x40024400)};
port_t PortF={0x1F,
  (unsigned long volatile *)(0x400253FC),
  (unsigned long volatile *)(0x40025400)};
int MakeOutput(port_t *ppt){
  (*ppt->ddr) = ppt->mask; // make output
  return 1;}
int MakeInput(port_t *ppt){
  (*ppt->ddr )= 0x00; // make input
  return 1;}
unsigned char Input( port_t *ppt){
 return (*ppt->addr);}
void Output(port_t *ppt, unsigned char data){
 (*ppt->addr) = data;
}
int main(void){ unsigned char MyData;
  MakeInput(&PortE);
  MakeOutput(&PortF);
  Output(&PortF,0);
  MyData=Input(&PortE);
  return 1;}
```

Listing 9-5: Port access organized with a data structure

Linear Linked Lists

One of the applications of structures involves linking elements together with pointers. A linear linked list is a simple 1-D data structure where the nodes are chained together one after another. Each node contains data and a link to the next node. The first node is pointed to by the **HeadPt** and the last node has a null-pointer in the next field. A node could be defined as

```
  struct node{
    unsigned short data;  // 16 bit information
    struct node *next;    // pointer to the next
  };
  typedef struct node node_t;
  node_t *HeadPt;
```

Listing 9-8: Linear linked list node structure

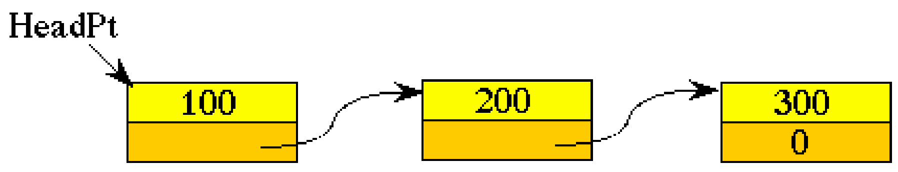


Figure 9-3: Linear linked list with 3 nodes

In order to store more data in the structure, we will first create a new node then link it into the list. The routine **StoreData** will return a true value if successful.

```
  #include <stdlib.h>;
  int StoreData(unsigned short info){ node_t *pt;
    pt=malloc(sizeof(node_t));  // create a new entry
    if(pt){
      pt->data=info;              // store data
      pt->next=HeadPt;            // link into existing
      HeadPt=pt;
      return(1);
    }
    return(0);      // out of memory
  };
```

Listing 9-9: Code to add a node at the beginning of a linear linked list

In order to search the list we start at the **HeadPt**, and stop when the pointer becomes **null**. The routine **Search** will return a pointer to the node if found, and it will return a null-pointer if the data is not found.

```
node_t *Search(unsigned short info){node_t *pt;
  pt=HeadPt;
  while(pt){
    if(pt->data==info)
        return (pt);
    pt=pt->next;   // link to next
  }
  return(pt);      // not found
};
```

Listing 9-10: Code to find a node in a linear linked list

To count the number of elements, we again start at the **HeadPt**, and stop when the pointer becomes **null**. The routine **Count** will return the number of elements in the list.

```
unsigned short Count(void){ node_t *pt;
  unsigned short cnt;
  cnt = 0;
  pt = HeadPt;
  while(pt){
    cnt++;
    pt = pt->next;   // link to next
  }
  return(cnt);
};
```

Listing 9-11: Code to count the number of nodes in a linear linked list

If we wanted to maintain a sorted list, then we can insert new data at the proper place, in between data elements smaller and larger than the one we are inserting. In the following figure we are inserting the element 250 in between elements 200 and 300.

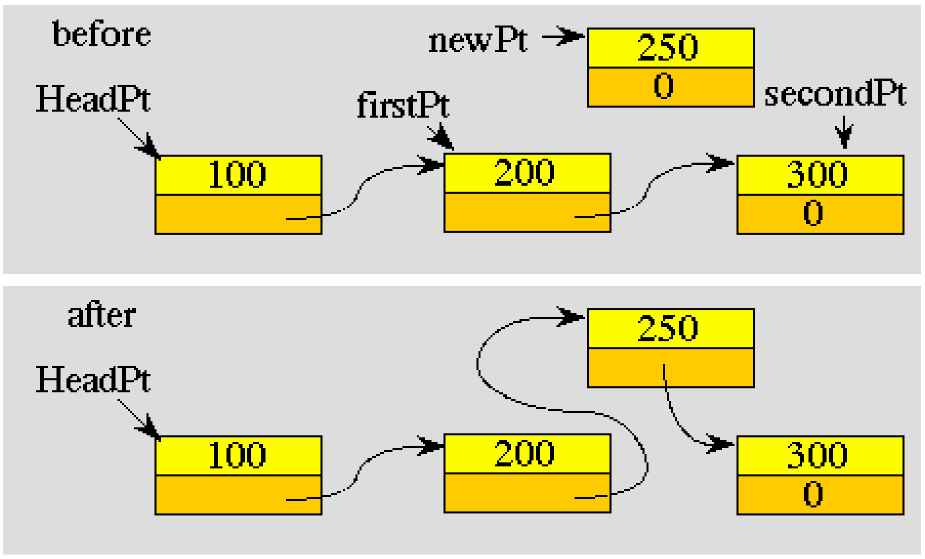


Figure 9-4: Inserting a node in sorted order

In case 1, the list is initially empty, and this new element is the first and only one. In case 2, the new element is inserted at the front of the list because it has the smallest data value. Case 3 is the general case depicted in the above figure. In this situation, the new element is placed in between

firstPt and **secondPt**. In case 4, the new element is placed at the end of the list because it has the largest data value.

```
int InsertData(unsigned short info){
node_t *firstPt,*secondPt,*newPt;
  newPt = malloc(sizeof(node_t));  // create a new entry
  if(newPt){
    newPt->data = info;              // store data
    if(HeadPt==0){   // case 1
      newPt->next = HeadPt;   // only element
      HeadPt = newPt;
      return(1);
    }
    if(info<=HeadPt->data){ // case 2
      newPt->next = HeadPt;   // first element in list
      HeadPt = newPt;
      return(1);
    }
    firstPt = HeadPt;   // search from beginning
    secondPt = HeadPt->next;
    while(secondPt){
      if(info <= secondPt->data){ // case 3
        newPt->next = secondPt;   // insert element here
        firstPt->next = newPt;
        return(1);
      }
      firstPt = secondPt;   // search next
      secondPt = secondPt->next;
    }
    newPt->next = secondPt;   // case 4, insert at end
    firstPt->next = newPt;
    return(1);
  }
  return(0);      // out of memory
};
```

Listing 9-12: Code to insert a node in a sorted linear linked list

The following function will search and remove a node from the linked list. Case 1 is the situation in which an attempt is made to remove an element from an empty list. The return value of zero signifies the attempt failed. In case 2, the first element is removed. In this situation the **HeadPt** must be updated to now point to the second element. It is possible the second element does not exist, because the list orginally had only one element. This is OK because in this situation **HeadPt** will be set to null signifying the list is now empty. Case 3 is the general situation in which the element at **secondPt** is removed. The element before, **firstPt**, is now linked to the element after. Case 4 is the situation where the element that was requested to be removed did not exist. In this case, the return value of zero signifies the request failed.

```
int Remove(unsigned short info){
node_t *firstPt,*secondPt;
  if(HeadPt==0)  // case 1
    return(0);   // empty list
  firstPt = HeadPt;
  secondPt = HeadPt->next;
  if(info==HeadPt->data){  // case 2
    HeadPt = secondPt; // remove first element in list
    free(firstPt);     // return unneeded memory to heap
    return(1);
  }
  while(secondPt){
    if(secondPt->data==info){  // case 3
      firstPt->next=secondPt->next; // remove this one
```

```
            free(secondPt);   // return unneeded memory to heap
            return(1);
          }
          firstPt = secondPt;   // search next
          secondPt = secondPt->next;
        }
        return(0);    // case 4, not found
      };
```

Listing 9-13: Code to remove a node from a sorted linear linked list

Example of a Huffman Code

When information is stored or transmitted there is a fixed cost for each bit. Data compression and decompression provide a means to reduce this cost without loss of information. If the sending computer compresses a message before transmission and the receiving computer decompresses it at the destination, the effective bandwidth is increased. In particular, this example introduces a way to process bit streams using Huffman encoding and decoding. A typical application is illustrated by the following flow diagram.

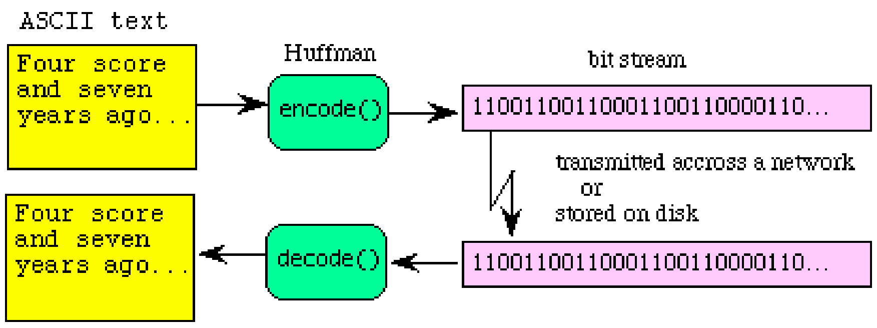


Figure 9-5: Data flow diagram showing a typical application of Huffman encoding and decoding

The Huffman code is similar to the Morse code in that they both use short patterns for letters that occur more frequently. In regular ASCII, all characters are encoded with the same number of bits (8). Conversely, with the Huffman code, we assign codes where the number of bits to encode each letter varies. In this way, we can use short codes for letter like "e s i a t o n" (that have a higher probability of occurrence) and long codes for seldom used consonants like "j q w z" (that have a lower probability of occurrence). To illustrate the encode-decode operations, consider the following Huffman code for the letters M,I,P,S. S is encoded as "0", I as "10", P as "110" and M as "111". We can store a Huffman code as a binary tree.

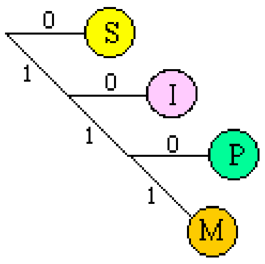


Figure 9-6: Huffman code for the letters S I P M

If "MISSISSIPPI" were to be stored in ASCII, it would require 10 bytes or 80 bits. With this simple Huffman code, the same string can be stored in 21 bits.

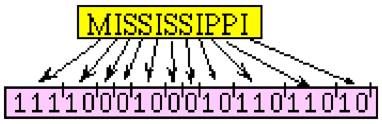


Figure 9-7: Huffman encoding for MISSISSIPPI

Of course, this Huffman code can only handle 4 letters, while the ASCII code has 128 possibilities, so it is not fair to claim we have an 80 to 21 bit savings. Nevertheless, for information that has a wide range of individual probabilities of occurrence, a Huffman code will be efficient. In the following implementation the functions **BitPut()** and **BitGet()** are called to save and recover binary data. The implementations of these two functions were presented back in Chapter 2.

```
struct Node{
    char Letter0;     // ASCII code if binary 0
    char Letter1;     // ASCII code if binary 1
// Letter1 is NULL(0) if Link is pointer to another node
    const struct Node *Link;};  // binary tree pointer
typedef const struct Node node_t;
// Huffman tree
node_t twentysixth= {'Q','Z',0};
node_t twentyfifth= {'X',0,&twentysixth};
node_t twentyfourth={'G',0,&twentyfifth};
node_t twentythird= {'J',0,&twentyfourth};
node_t twentysecond={'W',0,&twentythird};
node_t twentyfirst= {'V',0,&twentysecond};
node_t twentyth=    {'H',0,&twentyfirst};
node_t ninteenth=   {'F',0,&twentyth};
node_t eighteenth=  {'B',0,&ninteenth};
node_t seventeenth= {'K',0,&eighteenth};
node_t sixteenth=   {'D',0,&seventeenth};
node_t fifteenth=   {'P',0,&sixteenth};
node_t fourteenth=  {'M',0,&fifteenth};
node_t thirteenth=  {'Y',0,&fourteenth};
node_t twelfth=     {'L',0,&thirteenth};
node_t eleventh=    {'U',0,&twelfth};
node_t tenth=       {'R',0,&eleventh};
node_t ninth=       {'C',0,&tenth};
node_t eighth=      {'O',0,&ninth};
node_t seventh=     {' ',0,&eighth};
node_t sixth=       {'N',0,&seventh};
node_t fifth=       {'I',0,&sixth};
node_t fourth=      {'S',0,&fifth};
node_t third=       {'T',0,&fourth};
node_t second=      {'A',0,&third};
node_t root=        {'E',0,&second};
//********encode***************
// convert ASCII string to Huffman bit sequence
// returns bit count if OK
// returns 0         if BitFifo Full
// returns 0xFFFF    if illegal character
int encode(char *sPt){  // null-terminated ASCII string
 int NotFound; char data;
 int BitCount=0;       // number of bits created
 node_t *hpt;          // pointer into Huffman tree
 while (data=(*sPt)){
   sPt++;              // next character
   hpt=&root;          // start search at root
```

```
          NotFound=1;         // changes to 0 when found
          while(NotFound){
            if ((hpt->Letter0)==data){
              if(!BitPut(0))
                return (0);   // data structure full
              BitCount++;
              NotFound=0; }
            else {
              if(!BitPut(1))
                return (0);   // data structure full
              BitCount++;
             if ((hpt->Letter1)==data)
                NotFound=0;
              else {      // doesn't match either Letter0 or Letter1
                hpt=hpt->Link;
                if (hpt==0) return (0xFFFF); // illegal, end of tree?
              }
            }
          }
         }
         return BitCount;
        }
       //*******decode***************
       // convert Huffman bit sequence to ASCII
       // will remove from the BitFifo until it is empty
       // returns character count
       int decode(char *sPt){  // null-terminated ASCII string
        int CharCount=0;       // number of ASCII characters created
        unsigned int data;
        node_t *hpt;           // pointer into Huffman tree
        hpt=&root;             // start search at root
        while (BitGet(&data)){
          if (data==0){
            (*sPt)= (hpt->Letter0);
            sPt++;
            CharCount++;
            hpt=&root;}      // start over and search at root
          else  //data is 1
            if((hpt->Link)==0){
              (*sPt)= (hpt->Letter1);
              sPt++;
              CharCount++;
              hpt=&root;}    // start over and search at root
            else {       // doesn't match either Letter0 or Letter1
              hpt=hpt->Link;
            }
          }
          (*sPt)=0;  // null terminated
          return CharCount;
        }
```

Listing 9-14: A Huffman code implementation

Go to Chapter 10 on Functions Return to Table of Contents

Chapter 10: Functions

What's in Chapter 10?

We have been using functions throughout this document, but have put off formal presentation until now because of their immense importance. The key to effective software development is the appropriate division of a complex problem in modules. A module is a software task that takes inputs, operates in a well-defined way to create outputs. In C, functions are our way to create modules. A small module may be a single function. A medium-sized module may consist of a group of functions together with global data structures, collected in a single file. A large module may include multiple medium-sized modules. A hierarchical software system combines these software modules in either a top-down or bottom-up fashion. We can consider the following criteria when we decompose a software system into modules:

1) We wish to make the overall software system easy to understand;
2) We wish to minimize the coupling or interactions between modules;
3) We wish to group together I/O port accesses to similar devices;
4) We wish to minimize the size (maximize the number) of modules;
5) Modules should be able to be tested independently;
6) We should be able to replace/upgrade one module with effecting the others;
7) We would like to reuse modules in other situations.

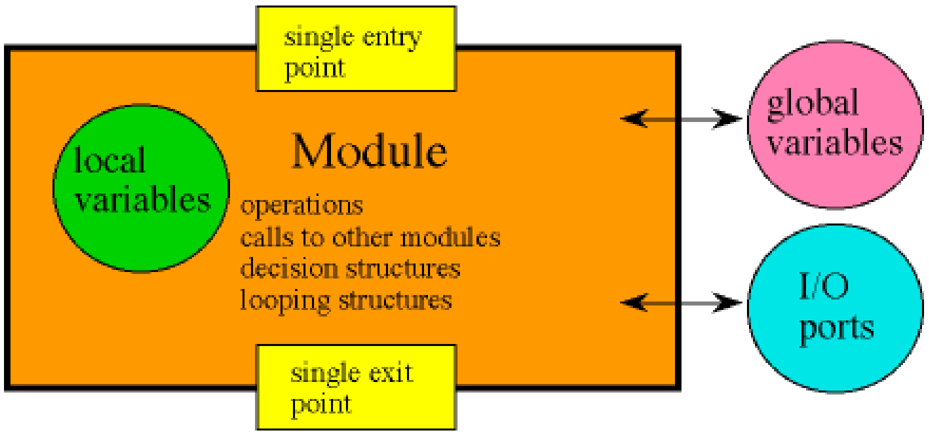


Figure 10-1: A module has inputs and outputs

As a programmer we must take special case when dealing with global variables and I/O ports. In order to reduce the complexity of the software we will limit access to global variables and I/O ports. It is essential to divide a large software task into smaller, well-defined and easy to debug modules. For more information about modular programming see either Chapter 5 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano, or Chapter 3 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

The term *function* in C is based on the concept of mathematical functions. In particular, a mathematical function is a well-defined operation that translates a set of input values into a set of output values. In C, a function translates a set of input values into a single output value. We will develop ways for our C functions to return multiple output values and for a parameter to be both an input and an output parameter. As a simple example consider the function that converts temperature in degrees F into temperature in degrees C.

```
short FtoC(short TempF){
     short TempC;
     TempC=(5*(TempF-32))/9;   // conversion
return TempC;}
```

When the function's name is written in an expression, together with the values it needs, it represents the result that it produces. In other words, an operand in an expression may be written as a function name together with a set of values upon which the function operates. The resulting value, as determined by the function, replaces the function reference in the expression. For example, in the expression

```
FtoC(T+2)+4;    // T+2 degrees Fahrenheit plus 4 degrees Centigrade
```

the term **FtoC(T+2)** names the function **FtoC** and supplies the variable **T** and the constant **2** from which **FtoC** derives a value, which is then added to **4**. The expression effectively becomes

```
((5*((T+2)-32))/9)+4;
```

Although **FtoC(T+2)+4** returns the same result as **((5*((T+2)-32))/9)+4**, they are not identical. As will we see later in this chapter, the function call requires the parameter **(T+2)** to be passed on the stack and a subroutine call will be executed.

Function Declarations

Similar to the approach with variables, C differentiates between a function declaration and a function definition. A declaration specifies the syntax (name and input/output parameters), whereas a function definition specifies the actual program to be executed when the function is called. Many C programmers refer to function declaration as a prototype. Since the C compiler is essential a one-pass process (not including the preprocessor), a function must be declared (or defined) before it can be called. A function declaration begins with the type (format) of the return parameter. If there is no return parameter, then the type can be either specified as **void** or left blank. Next comes the function name, followed by the parameter list. In a function declaration we do not have to specify names for the input parameters, just their types. If there are no input parameters, then the type can be either specified as **void** or left blank. The following examples illustrate that the function declaration specifies the name of the function and the types of the function parameters.

```
//  declaration                     input            output
void Ritual(void);               // none             none
char InChar(void);               // none             8-bit
void OutChar(char);              // 8-bit            none
short InSDec(void);              // none             16-bit
void OutSDec(short);             // 16-bit           none
char Max(char,char);             // two 8-bit        8-bit
int EMax(int,int);               // two 32-bit       32-bit
void OutString(char*);           // pointer to 8-bit none
char *alloc(int);                // 32-bit           pointer to 8-bit
int Exec(void(*fnctPt)(void));  // function pointer 32-bit
```

Normally we place function declarations in the header file. We should add comments that explain what the function does.

```
void InitSCI(void);   // Initialize 38400 bits/sec
char InChar(void);    // Reads in a character, gadfly
void OutChar(char);   // Output a character, gadfly
char UpCase(char);    // Converts lower case character to upper case
void InString(char *, unsigned int); // Reads in a String of max length
```

To illustrate some options when declaring functions, we give alternative declarations of these same five functions:

```
InitSCI();
char InChar();
void OutChar(char letter);
char UpCase(char letter);
InString(char *pt, unsigned int MaxSize);
```

Sometimes we wish to call a function that will be defined in another module. If we define a function as external, software in this file can call the function (because the compiler knows everything about the function except where it is), and the linker will resolve the unknown address later when the object codes are linked.

```
extern void InitSCI(void);
extern char InChar(void);
extern void OutChar(char);
extern char UpCase(char);
extern void InString(char *, unsigned int);
```

One of the power features of C is to define pointers to functions. A simple example follows:

```
int (*fp)(int);  // pointer to a function with input and output
int fun1(int input){
   return(input+1);    // this adds 1
};
int fun2(int input){
   return(input+2);    // this adds 2
};
void Setp(void){ int data;
   fp = &fun1;      // fp points to fun1
   data = (*fp)(5); // data=fun1(5);
   fp = &fun2;      // fp points to fun2
   data = (*fp)(5); // data=fun2(5);
};
```

Listing 10-1: Example of a function pointer

The declaration of **fp** looks a bit complicated because it has two sets of parentheses and an asterisk. In fact, it declares **fp** to be a pointer to any function that returns integers. In other words, the line **int (*fp)(int);** doesn't define the function. As in other declarations, the asterisk identifies the following name as a pointer. Therefore, this declaration reads "**fp** is a pointer to a function with a 32-bit signed input parameter that returns a 32-bit signed output parameter." Using the term *object* loosely, the asterisk may be read in its usual way as "object at." Thus we could also read this declaration as "the object at **fp** is a function with an **int** input that returns an **int**."

So why the first set of parentheses? By now you have noticed that in C declarations follow the same syntax as references to the declared objects. And, since the asterisk and parentheses (after the name) are expression operators, an evaluation precedence is associated with them. In C, parentheses following a name are associated with the name before the preceding asterisk is applied to the result. Therefore,

```
int *fp(int);
```

would be taken as

```
int *(fp(int));
```

saying that **fp** is a function returning a pointer to an integer, which is not at all like the declaration in Listing 10-1.

Function Definitions

The second way to declare a function is to fully describe it; that is, to *define* it. Obviously every function must be defined somewhere. So if we organize our source code in a bottom up fashion, we would place the lowest level functions first, followed by the function that calls these low level functions. It is possible to define large project in C without ever using a standard declaration (function prototype). On the other hand, most programmers like the top-down approach illustrated in the following example. This example includes three modules: the LCD interface, the UART functions, and some SysTick timer routines. Notice the function names are chosen to reflect the module in which they are defined. If you are a C++ programmer, consider the similarities between this C function call **LCD_clear()** and a C++ LCD class and a call to a member function **LCD.clear()**. The *.H files contain function declarations and the *.C files contain the implementations.

```
#include "LCD.h"
#include "UART.H"
#include "SysTick.H"
void main(void){ char letter; short n=0;
   UART_Init();
   LCD_Init();
   SysTick_Init()
   LCD_String("This is a LCD");
   SysTick_Wait10ms(1000);
   LCD_clear();
   letter='a'-1;
   while(1){
      if (letter=='z')
         letter='a';
      else
         letter++;
      LCD_putchar(letter);
      SysTick_Wait10ms(250);
      if(++n==16){
         n=0;
         LCD_clear();
      }
   }
}
```

Listing 10-2: Modular approach to software development

C function definitions have the following form

type Name(*parameter list*){
CompoundStatement
};

Just like the function declaration, we begin the definition with its **type**. The **type** specifies the function return parameter. If there is no return parameter we can use **void** or leave it blank. **Name** is the name of the function. The **parameter list** is a list of zero or more names for the arguments that will be received by the function when it is called. Both the type and name of each input parameter is required. .

Although a character is passed in a 32-bit register, we are free to declare its formal argument as either character or word. If it is declared as a character, only the low-order byte of the actual argument will be referenced. If it is declared as an integer, then all 32 bits will be referenced.

It is generally more efficient to reference integers than characters because there is no need for a machine instruction to set the high-order byte. So it is common to see situations in which a character is passed to a function which declares the argument to be an integer. But there is one caveat here: not all C compilers promote character arguments to integers when passing them to functions; the result is an unpredictable value in the high-order byte of the argument. This should be remembered as a portability issue.

Since there is no way in C to declare strings, we cannot declare formal arguments as strings, but we can declare them as character pointers or arrays. In fact, as we have seen, C does not recognize strings, but arrays of characters. The string notation is merely a shorthand way of writing a constant array of characters.

Furthermore, since an unsubscripted array name yields the array's address and since arguments are passed by value, an array argument is effectively a pointer to the array. It follows that, the formal argument declarations **arg[]** and ***arg** are really equivalent. The compiler takes both as pointer declarations. Array dimensions in argument declarations are ignored by the compiler since the function has no control over the size of arrays whose addresses are passed to it. It must either assume an array's size, receive its size as another argument, or obtain it elsewhere.

The last, and most important, part of the function definition above is **CompoundStatement**. This is where the action occurs. Since compound statements may contain local declarations, simple statements, and other compound statements, it follows that functions may implement algorithms of any complexity and may be written in a structured style. Nesting of compound statements is permitted without limit.

As an example of a function definition consider

```
int add3(int z1, int z2, int z3){ int y;
    y=z1+z2+z3;
    return(y);}
```

Listing 10-3: Example function with 3 inputs and one output.

Here is a function named **add3** which takes three input arguments.

Function Calls

A function is called by writing its name followed by a parenthesized list of argument expressions. The general form is

Name **(***parameter list***)**

where **Name** is the name of the function to be called. The **parameter list** specifies the particular input parameters used in this call. Notice that each input parameter is in fact an expression. It may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. Whatever the case, the resulting value is pushed onto the stack where it is passed to the called function.

C programs evaluate arguments from left to right, pushing them onto the stack in that order. On return, the return parameter is located in Reg R0. The input parameters are removed from the stack at the end of the program.

When the called function receives control, it refers to the first actual argument using the name of the first formal argument. The second formal argument refers to the second actual argument, and so on. In other words, actual and formal arguments are matched by position in their respective lists. Extreme care must be taken to ensure that these lists have the same number and type of arguments.

It was mentioned earlier, that function calls appear in expressions. But, since expressions are legal statements, and since expressions may consist of only a function call, it follows that a function call may be written as a complete statement. Thus the statement

```
add3(--counter,time+5,3);
```

is legal. It calls **add3()**, passing it three arguments **--counter**, **time+5**, and **3**. Since this call is not part of a larger expression, the value that **add3()** returns will be ignored. As a better example, consider

```
y=add3(--counter,time+5,3);
```

which is also an expression. It calls **add3()** with the same arguments as before but this time it assigns the returned value to **y**. It is a mistake to use an assignment statement like the above with a function that does not return an output parameter.

The ability to pass one function a pointer to another function is a very powerful feature of the C language. It enables a function to call any of several other functions with the caller determining which subordinate function is to be called.

```
int fun1(int input){
   return(input+1);    // this adds 1
};
int fun2(int input){
   return(input+2);    // this adds 2
};
int execute(int (*fp)(int)){ int data;
   data = (*fp)(5); // data=fun1(5);
   return (data);
};
void main(void){ int result;
   result = execute(&fun1); // result=fun1(5);
   result = execute(&fun2); // result=fun2(5);
};
```

Listing 10-4: Example of passing a function pointer

Notice that **fp** is declared to be a function pointer. Also, notice that the designated function is called by writing an expression of the same form as the declaration.

Argument Passing

Now let us take a closer look at the matter of argument passing. With respect to the method by which arguments are passed, two types of subroutine calls are used in programming languages--

call by reference and *call by value*.

The *call by reference* method passes arguments in such a way that references to the formal arguments become, in effect, references to the actual arguments. In other words, references (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. In this scheme, assignment statements have implied side effects on the actual arguments; that is, variables passed to a function are affected by changes to the formal arguments. Sometimes side effects are beneficial, and some times they are not. Since C supports only one formal output parameter, we can implement additional output parameters using call by reference. In this way the function can return parameters back using the reference. As an example recall the fifo queue program shown earlier in Listing 8-7. The function **GetFifo**, shown below, returns two parameters. The regular formal parameter is a boolean specifying whether or not the request was successful, and the actual data removed from the queue is returned via the call by reference. The calling program **InChar** passes the address of its local variable data. The assignment statement ***datapt=Fifo[GetI++];** within **GetFifo** will store the return parameter into a local variable of **InChar**. Normally **GetFifo** does not have the scope to access local variables of **InChar**, but in this case **InChar** explicitly granted that right by passing a pointer to **GetFifo**.

```
int GetFifo (char *datapt) {
  if(Size == 0 )
    return(0);     /* Empty if Size=0 */
  else{
    *datapt=Fifo[GetI++]; Size--;
    if (GetI == FifoSize) GetI = 0;
    return(-1);
  }
}
char InChar(void){ char data;
  while(GetFifo(&data)){};
  return (data);}
```

Listing 10-5: Multiple output parameters can be implemented using call by reference

When we use the *call by value* scheme, the values, not references, are passed to functions. With call by value copies are made of the parameters. Within a called function, references to formal arguments see copied values on the stack, instead of the original objects from which they were taken. At the time when the computer is executing within PutFifo() of the example below, there will be three separate and distinct copies of the 0x41 data (main, OutChar and PutFifo).

```
int PutFifo(char data) {
  if(Size == FifoSize ) {
     return(0);} /* Failed, fifo was full */
  else{
    Size++;
    *(PutPt++)=data; /* put data into fifo */
    if (PutPt == &Fifo[FifoSize]) PutPt = &Fifo[0]; /* Wrap */
    return(-1); /* Successful */
  }
}
void OutChar(char data){
   while(PutFifo(data)){};
}
void main(void){ char data=0x41;
   OutChar(data);
}
```

Listing 10-6: Call by value passes a copy of the data.

The most important point to remember about passing arguments by value in C is that there is no connection between an actual argument and its source. Changes to the arguments made within a function, have no affect what so ever on the objects that might have supplied their values. They can be changed with abandon and their sources will not be affected in any way. This removes a burden of concern from the programmer since he may use arguments as local variables without side effects. It also avoids the need to define temporary variables just to prevent side effects.

It is precisely because C uses call by value that we can pass expressions, not just variables, as arguments. The value of an expression can be copied, but it cannot be referenced since it has no existence in global memory. Therefore, call by value adds important generality to the language.

Although the C language uses the call by value technique, it is still possible to write functions that have side effects; but it must be done deliberately. This is possible because of C's ability to handle expressions that yield addresses. And, since any expression is a valid argument, addresses can be passed to functions.

Since expressions may include assignment, increment, and decrement operators (Chapter 9), it is possible for argument expressions to affect the values of arguments lying to their right. (Recall that C evaluates argument expressions from left to right.) Consider, for example,

```
func (y=x+1, 2*y);
```

where the first argument has the value **x+1** and the second argument has the value **2*(x+1)**. What would be the value of the second argument if arguments were evaluated from right to left? This kind of situation should be avoided, since the C language does not guarantee the order of argument evaluation. The safe way to write this is

```
y=x+1;
func (y, 2*y);
```

It is the programmer's responsibility to ensure that the parameters passed match the formal arguments in the function's definition. Some mistakes will be caught as syntax errors by the compiler, but this mistake is a common and troublesome problem for all C programmers.

Occasionally, the need arises to write functions that work with a variable number of arguments. An example is **printf()** in the library. In C, this feature is implemented using macros defined in the library file STDARG.C. To use these features you include STDARG.H in your file.

Private versus Public Functions

For every function definition, the compiler generates an assembler directive declaring the function's name to be *public*. This means that every C function is a potential entry point and so can be accessed externally. One way to create private/public functions is to control which functions have declarations. Consider again the main program in Listing 10-2 shown earlier. Now lets look inside the Timer.H and Timer.C files. To implement Private and Public functions we place the function declarations of the Public functions in the Timer.H file.

```
void SysTick_Init(void);
void SysTick_Wait(unsigned long delay);
```

Listing 10-7: SysTick.h header file has public functions

The implementations of all functions are included in the SysTick.c file. The function, **SysTick_Wait**, is private and can only be called by software inside the SysTick.c file. We can apply this same approach to private and public global variables. Notice that in this case the global variable, **TimerClock**, is private and can not be accessed by software outside the SysTick.c file.

```
unsigned long static TimerClock; // private global
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // 1) disable SysTick during setup
  NVIC_ST_RELOAD_R = 0x00FFFFFF;    // 2) maximum reload value
  NVIC_ST_CURRENT_R = 0;            // 3) any write to current clears it
  NVIC_ST_CTRL_R = 0x00000005;      // 4) enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void static SysTick_Wait(unsigned long delay){
  NVIC_ST_RELOAD_R = delay-1;  // number of counts to wait
  NVIC_ST_CURRENT_R = 0;       // any value written to CURRENT clears
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
  }
} // 10000us equals 10ms
void SysTick_Wait10ms(unsigned long delay){
  unsigned long i;
  for(i=0; i<delay; i++){
    SysTick_Wait(800000);  // wait 10ms
  }
}
```

Listing 10-8: Timer.C implementation file defines all functions

For more information about modular programming see either Chapter 5 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano, or Chapter 3 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

Finite State Machine using Function Pointers

Now that we have learned how to declare, initialize and access function pointers, we can create very flexible finite state machines. In the finite state machine presented in Listing 9-2 and Listing 9-4, the output was a simple number that is written to the output port. In this next example, we will actually implement the exact same machine, but in a way that supports much more flexibility in the operations that each state performs. In fact we will define a general C function to be executed at each state. In this implementation the functions perform the same output as the previous machine.

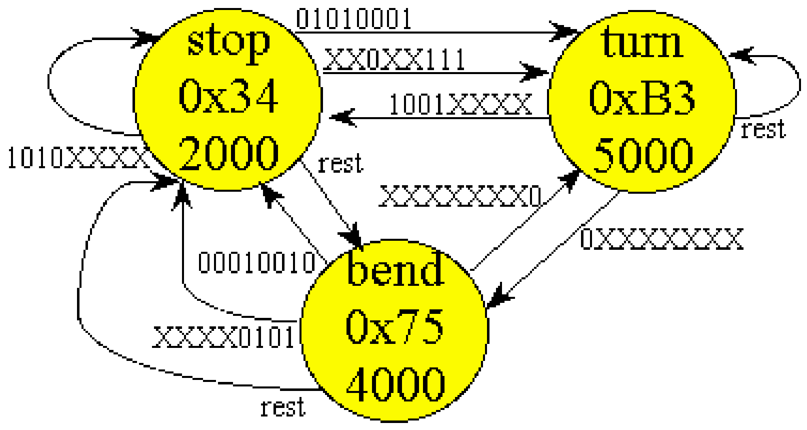


Figure 10-6: Finite State Machine (same as Figure 9-1)

Compare the following implementation to Listing 9-2, and see that the **unsigned char Out;** constant is replaced with a **void (*CmdPt)(void);** function pointer. The three general function **DoStop() DoTurn()** and **DoBend()** are also added.

```
struct State{
    void (*CmdPt)(void);      /* function to execute */
    unsigned short Wait;     /* Time, 10ms  to wait */
    unsigned char AndMask[4];
    unsigned char EquMask[4];
    const struct State *Next[4];};  /* Next states */
typedef const struct State state_t;
typedef state_t * StatePtr;
#define stop &fsm[0]
#define turn &fsm[1]
#define bend &fsm[2]
void DoStop(void){  PORTA = 0x34;}
void DoTurn(void){  PORTA = 0xB3;}
void DoBend(void){  PORTA = 0x75;}
state_t fsm[3]={
{&DoStop, 2000,   // stop 1 ms
   {0xFF,   0xF0,   0x27,   0x00},
   {0x51,   0xA0,   0x07,   0x00},
   {turn,   stop,   turn,   bend}},
{&DoTurn,5000,   // turn 2.5 ms
   {0x80,   0xF0,   0x00,   0x00},
   {0x00,   0x90,   0x00,   0x00},
   {bend,   stop,   turn,   turn}},
{&DoBend,4000,   // bend 2 ms
   {0xFF,   0x0F,   0x01,   0x00},
   {0x12,   0x05,   0x00,   0x00},
   {stop,   stop,   turn,   stop}}};
```

Listing 10-12: Linked finite state machine structure stored in ROM

Compare the following implementation to Listing 9-4, and see that the **PORTH=pt-Out;** assignment is replaced with a **(*Pt->CmdPt)();** function call. In this way, the appropriate function **DoStop() DoTurn()** or **DoBend()** will be called.

```
void control(void){ StatePtr Pt;
  unsigned char Input; unsigned short startTime; unsigned int i;
  SysTick_Init();
  Port_Init();

  Pt = stop;       // Initial State
  while(1){
    (*Pt->CmdPt)();         // 1) execute function
    SysTick_Wait10ms(Pt->Wait);   // 2) wait
    Input = PORTB;          // 3) input
    for(i=0;i<4;i++)
      if((Input&Pt->AndMask[i])==Pt->EquMask[i]){
        Pt=Pt->Next[i]; // 4) next depends on input
        i=4; }}};
```

Listing 10-13: Finite state machine controller

Linked List Interpreter using Function Pointers

In the next example, function pointers are stored in a listed-list. An interpreter accepts ASCII input from a keyboard and scans the list for a match. In this implementation, each node in the linked list has a function to be executed when the operator types the corresponding letter. The linked list **LL** has three nodes. Each node has a letter, a function and a link to the next node.

```
// Linked List Interpreter
struct Node{
    unsigned char Letter;
    void (*fnctPt)(void);
    const struct Node *Next;};
typedef const struct Node node_t;
typedef node_t * NodePtr;
void CommandA(void){
    OutString("\nExecuting Command a");
}
void CommandB(void){
    OutString("\nExecuting Command b");
}
void CommandC(void){
    OutString("\nExecuting Command c");
}
node_t LL[3]={
    { 'a', &CommandA, &LL[1]},
    { 'b', &CommandB, &LL[2]},
    { 'c', &CommandC, 0 }};
void main(void){ NodePtr Pt; char string[40];
    UART_Init(); // Enable SCI port
    UART_OutString("\nEnter a single letter command followed by <enter>");
    while(1){
        UART_OutString("\n>");
        UART_InString(string,39); // first character is interpreted
        Pt=&LL[0]; // first node to check
        while(Pt){
            if(string[0]==Pt->Letter){
                Pt->fnctPt(); // execute function
                break;}       // leave while loop
            else{
                Pt=Pt->Next;
                if(Pt==0) UART_OutString(" Error");}}}}
```

Listing 10-14: Linked list implementation of an interpreter.

Compare the syntax of the function call, **(*Pt->CmdPt)();**, in Listing 10-13, with the syntax in this example, **Pt->fnctPt();**.These two expressions both generate code that executes the function.

Go to Chapter 11 on Preprocessor Directives Return to Table of Contents

Chapter 11: Preprocessor Directives

What's in Chapter 11?

C compilers incorporate a preprocessing phase that alters the source code in various ways before passing it on for compiling. Four capabilities are provided by this facility in C. They are:

macro processing
inclusion of text from other files
conditional compiling
in-line assembly language

The preprocessor is controlled by directives which are not part of the C language proper. Each directive begins with a #character and is written on a line by itself. Only the preprocessor sees these directive lines since it deletes them from the code stream after processing them.

Depending on the compiler, the preprocessor may be a separate program or it may be integrated into the compiler itself. C has an integrated preprocessor that operates at the front end of its single pass algorithm.

Macro Processing

We use macros for three reasons. 1) To save time we can define a macro for long sequences that we will need to repeat many times. 2) To clarify the meaning of the software we can define a macro giving a symbolic name to a hard-to-understand sequence. The I/O port #define macros are good examples of this reason. 3) To make the software easy to change, we can define a macro such that changing the macro definition, automatically updates the entire software.

```
#define Name CharacterString?...
```

define names which stand for arbitrary strings of text. After such a definition, the preprocessor replaces each occurrence of *Name* (except in string constants and character constants) in the source text with *CharacterString?...*. As C implements this facility, the term macro is misleading, since parameterized substitutions are not supported. That is, *CharacterString?...* does not change from one substitution to another according to parameters provided with *Name* in the source text.

C accepts macro definitions only at the global level.

The *Name* part of a macro definition must conform to the standard C naming conventions as described in Chapter 2. *CharacterString?...* begins with the first printable character following *Name* and continues through the last printable character of the line or until a comment is reached.

If *CharacterString?...* is missing, occurrences of *Name* are simply squeezed out of the text. Name matching is based on the whole name (up to 8 characters); part of a name will not match. Thus the directive

```
#define size 10
```

will change

```
    short data[size];
```

into

```
    short data[10];
```

but it will have no effect on

```
    short data[size1];
```

Replacement is also performed on subsequent **#define** directives, so that new symbols may be defined in terms of preceding ones.

The most common use of **#define** directives is to give meaningful names to constants; i.e., to define so called *manifest constants*. However, we may replace a name with anything at all, a commonly occurring expression or sequence of statements for instance. `TriggerPendSV()` will set bit 20, triggering a PendSV. `SetPA5(0x20` will set PA5. `Wait(500)` will execute the loop 500 times.

```
    #define TriggerPendSV() { NVIC_INT_CTRL_R = 0x10000000;}
    #define SetPA5(x) {GPIO_PORTA_DATA_R = (GPIO_PORTA_DATA_R & ~0x20) | (x)};
    #define Wait(t) {int wait; for (wait = 0; wait < (t); wait++) {}};
    #define CONVERT4BPP(c)  ( ((c) << 12) | ((c) << 7 ) | ((c) << 1) )
```

Listing 11.1: Example of #define

Conditional Compiling

This preprocessing feature lets us designate parts of a program which may or may not be compiled depending on whether or not certain symbols have been defined. In this way it is possible to write into a program optional features which are chosen for inclusion or exclusion by simply adding or removing **#define** directives at the beginning of the program.

When the preprocessor encounters

```
    #ifdef Name
```

it looks to see if the designated name has been defined. If not, it throws away the following source lines until it finds a matching

```
    #else
```

or

```
    #endif
```

directive. The **#endif** directive delimits the section of text controlled by **#ifdef**, and the **#else** directive permits us to split conditional text into true and false parts. The first part (**#ifdef...#else**) is compiled only if the designated name is defined, and the second (**#else...#endif**) only if it is not defined.

The converse of **#ifdef** is the

```
#ifndef Name
```

directive. This directive also takes matching **#else** and **#endif** directives. In this case, however, if the designated name is not defined, then the first (**#ifndef...#else**) or only (**#ifndef...#endif**) section of text is compiled; otherwise, the second (**#else...#endif**), if present, is compiled.

Nesting of these directives is allowed; and there is no limit on the depth of nesting. It is possible, for instance, to write something like

```
#ifdef ABC
... /* ABC */
#ifndef DEF
... /* ABC and not DEF */
#else
... /* ABC and DEF */
#endif
... /* ABC */
#else
... /* not ABC */
#ifdef HIJ
... /* not ABC but HIJ */
#endif
... /* not ABC */
#endif
```

Listing 11.2: Examples on conditional compilation

where the ellipses represent conditionally compiled code, and the comments indicate the conditions under which the various sections of code are compiled.

A good application of conditional compilation is inserting debugging instrumemts. In this example the only purpose of writing to PORTC is assist in performance debugging. Once the system is debugged,we can remove all the debugging code, simply by deleting the `#define Debug 1` line.

```
#define Debug 1
int Sub(int j){ int i;
#ifdef Debug
    PORTC |= 0x80; /* PC7 set when Sub is entered */
#endif
    i=j+1;
#ifdef Debug
    PORTC &= ~0x80; /* PC7 cleared when Sub is exited */
#endif
    return(i);}
void Program(){ int i;
#ifdef Debug
    PORTC |=0x40; /* PC6 set when Program is entered */
#endif
    i=Sub(5);
    while(1) { PORTB=2; i=Sub(i);}}
void ProgB(){ int i;
    i=6;
#ifdef Debug
    PORTC &= ~0x40; /* PC6 cleared when Sub is exited */
#endif
}
```

Listing 11.3: Conditional compilation can help in removing all debugging code

Including Other Source Files

The preprocessor also recognizes directives to include source code from other files. The two directives

```
#include "Filename"

#include <Filename>
```

cause a designated file to be read as input to the compiler. The difference between these two directives is where the compiler looks for the file. The <filename> version will search for the file in the standard include directory, while the "filename" version will search for the file in the same directory as the original source file. The preprocessor replaces these directives with the contents of the designated files. When the files are exhausted, normal processing resumes.

Filename follows the normal MS-DOS file specification format, including drive, path, filename, and extension.

In Chapter 10, an example using #include was presented that implemented a feature similar to encapsulated objects of C++, including private and public functions.

Inline assembly

Keil uVision uses this syntax to embed assembly code into C programs

```
__asm void
Delay(unsigned long ulCount)
{
  subs    r0, #1
  bne     Delay
  bx      lr
}
```

Listing *11-4: Example of an assembly function*

Code composer studio implements assembly functions in a similar manner.

```
  void Delay(unsigned long ulCount){
  __asm (  "    subs    r0, #1\n"
      "    bne     Delay\n"
      "    bx      lr\n");
}
```

Listing 11-5: Example of an assembly function.

Return to Table of Contents

C Declarations: A Short Primer

Based on an article by Greg Comeau,

published in the September 1998 edition of the Microsoft Systems Journal

In the ariticle "A Guide to Understanding Even the Most Complex C Declarations", Greg Comeau presents a set of rules that can be applied to interpret any C declaration however complex it may seem. While the rules are intuitive and might appeal to most advanced C programmers, the beginner may find them difficult to grasp. He does however start the article by presenting a simple rule-set to read and write Kernighan and Ritchie (of the famous book, *The C Progamming Language, 1978*) style declarations. In this Primer I will present these rules and elaboarte on them with examples.

Here is the standard sytax for C Declarations:

The sytax of a C declaration is of the form:

```
        storage-class type qualifier declarator = initializer;
```

where `storage-class` is only one of the following:

```
     typedef
     extern
     static
     auto
     register
```

A `type` could be one or more of the following:

```
     void
     char
     short, int, long
     float, double
     signed, unsigned
     struct ...
     union ...
     typedef type
```

A `qualifier` could be one or more of the following:

```
     const
     volatile
```

A `declarator` contains an identifier and one or more, or none at all, of the following in a variety of combinations:

```
     *
     ()
     []
```

possibly grouped within parentheses to create different bindings

The term storage-class refers to the method by which an object is assigned space in memory. Chapter 4 of the C Primer gives detailed descriptions of what each of the storage-classes mean. Suffice it to say that understanding *what* is being declared has no bearing on the storage-class, as it specifically tells you *where* it is being declared and assigned space in memory. Also, qualifiers (const, volatile) refer respectively to the non-modifiability of an entity, and the fact that the entity in question is modified elsewhere. Therefore, henceforth we will ignore these two pieces of information.

The above definition simply says what a declaration ought to look like (the syntax that is). The key phrase in the above definition is "to create different bindings". What this means is, to give different interpretations to the declaration based on parenthesizing the declaration. All one has to understand any complex C declaration then, is to know that these declarations are based on the C operator precedence chart, the same one you use to evaluate expressions in C:

Precedence	Operators	Associativity
highest	() [] . -> ++(postfix) --(postfix)	left to right
	++(prefix) --(prefix) !~ **sizeof**(type) +(unary) -(unary) &(address) * (dereference)	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= <<= >>= |= &= ^=	right to left
lowest	,	left to right

This chart is complicated because it gives the precedence and associativity of all C operators. With declarations, we are only dealing with unary tokens (unary operators need only one operand) so it is a lot simpler The operators of interest to us are marked in red in the above table.

So, here then are the rules for reading and writing C declarations:

1. Parenthesize declarations as if they were expressions.
2. Locate the innermost parentheses.
3. Say "identifier is" where the identifier is the name of the variable.
   a. Say "an array of X" if you see [X].
   b. Say "a pointer to" if you see *.
   c. Say "A function returning" if you see ();
4. Move to the next set of parentheses.

5. If more, go back to 3.
6. Else, say "type" for the remaining type left (such as short int)

Here are some examples to clarify this process:

Example 1:
```
int i;
```
The parenthesization of the above declaration is:
```
int (i); {1}
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
No more parentheses left so we say "an int". {4,5,6}
That is, "i is an int"
```

Example 2:
```
int *i;
```
The parenthesization of the above declaration is:
```
int (*(i)); {1}
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: (*(i)) {4}
Go back to step 3. {5}
We say "a pointer to" since we see a * {3.b}
No more parentheses left so we say "an int". {4,5,6}
That is, "i is a pointer to an int"
```

Example 3:
```
int *i[3];
```
The parenthesization of the above declaration is:
```
int (*((i)[3])); {1} // Note that () and [] have the same
                     //   precedence but we deal with them from left to right.
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: ((i)[3]) {4}
Go back to step 3. {5}
We say "an array of 3 ..." since we see a [3] {3.a}
Move to the next set of parenthesis: (*((i)[3])) {4}
Go back to step 3. {5}
No more parentheses left so we say "ints". {4,5,6}
That is, "i is an array of 3 ints"
```

Example 4:
```
int (*i)[3];
```
The parenthesization of the above declaration is:
```
int ((*(i))[3]); {1} // Note that parentheses are valid tokens
                     //   in a declaration and therefore must be
                     //   be left in place when finding the final
                     //   parenthesization
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: (*(i)) {4}
Go back to step 3. {5}
We say "a pointer to ..." since we see a * {3.b}
Move to the next set of parenthesis: ((*(i))[3]) {4}
Go back to step 3. {5}
We say "an array of 3 ..." since we see a [3] {3.a}
No more parentheses left so we say "ints". {4,5,6}
That is, "i is a pointer to an array of 3 ints"
```

Example 5:
```
int *i();
```
The parenthesization of the above declaration is:
```
int (*((i)())); {1} // Note that * has a lower precedence than
                    //   parentheses, ()
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
// One could argue that () is also the innermost parenthesis but
//   it does not contain anything so we know it must indicate
//   a function
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: ((i)()) {4}
Go back to step 3. {5}
We say "a function returning" since we see a () {3.c}
Move to the next set of parenthesis:  (*((i)())) {4}
Go back to step 3. {5}
We say "a pointer to ..." since we see a * {3.b}
No more parentheses left so we say "an int". {4,5,6}
That is, "i is a function returning a pointer to an int"
```

Example 6:
```
int (*i)();
```
The parenthesization of the above declaration is:
```
int ((*(i))()); {1} // Note that parentheses are valid tokens
                    //   in a declaration and therefore must be
                    //   be left in place when finding the final
                    //   parenthesization
```
Applying the rules (see above) to the parenthesized expression can be done as follows:
```
The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: (*(i)) {4}
Go back to step 3. {5}
```

```
We say "a pointer to" since we see a * {3.b}
Move to the next set of parenthesis:  ((*(i))()) {4}
Go back to step 3. {5}
We say "a function returning" since we see a () {3.c}
No more parentheses left so we say "an int". {4,5,6}
That is, "i is a pointer to a function returning an int"
```

This is the pretty much all one needs to know to read and write declarations in C.