Lecture 2 – Embedded C

Program structure. Tokens. Numbers, characters and strings. Variables and constants. Expressions. Procedural statements. Pointers. Arrays and strings. Structures. Functions. Preprocessor directives.

This document gives a basic overview of programming in C for an embedded system.

Program Structure

Some basic terms will be introduced so that you get a feel for the language. It is not important yet that you understand the example programs fully. The examples are included to illustrate particular features of the language.

Case Study 1: 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.

A digital lock using a microcontroller

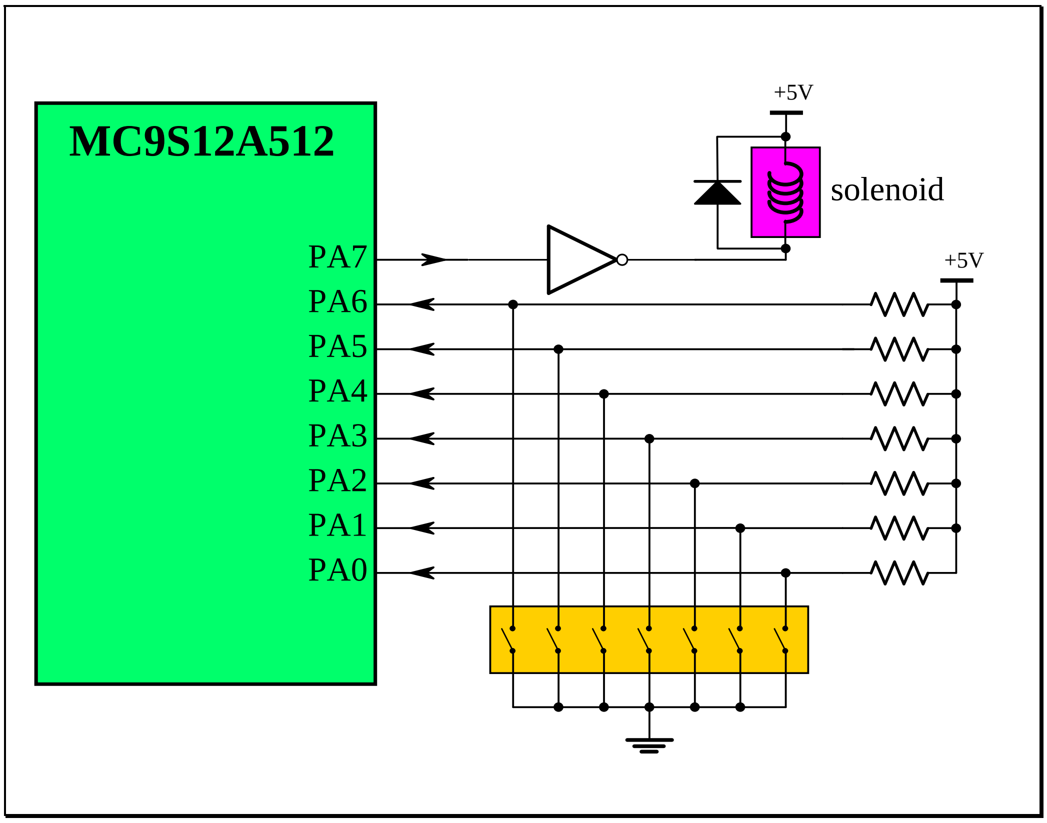


Figure 2.1 – Digital lock hardware

If the 7-bit binary pattern on Port A bits 6-0 becomes `0100011` for at least 10 ms, then the solenoid will activate. The 10 ms delay will compensate for the switch bounce. We see 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. The steps below are listed in a 1, 2, 3, … order, whereas in reality we iterate these steps over and over.

Iterative steps used to write software

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 symbolic names, `PORTA` and `DDRA`, to the corresponding addresses of the ports, `0x0000` and `0x0002`.

Accessing microcontroller ports in C

```
#define PORTA *(unsigned char volatile *)(0x0000)
#define DDRA *(unsigned char volatile *)(0x0002)
```

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 allocated in global space. If the software will change its value then it will be allocated in RAM. In this example we need a 16-bit unsigned counter.

```
unsigned int Count;
```

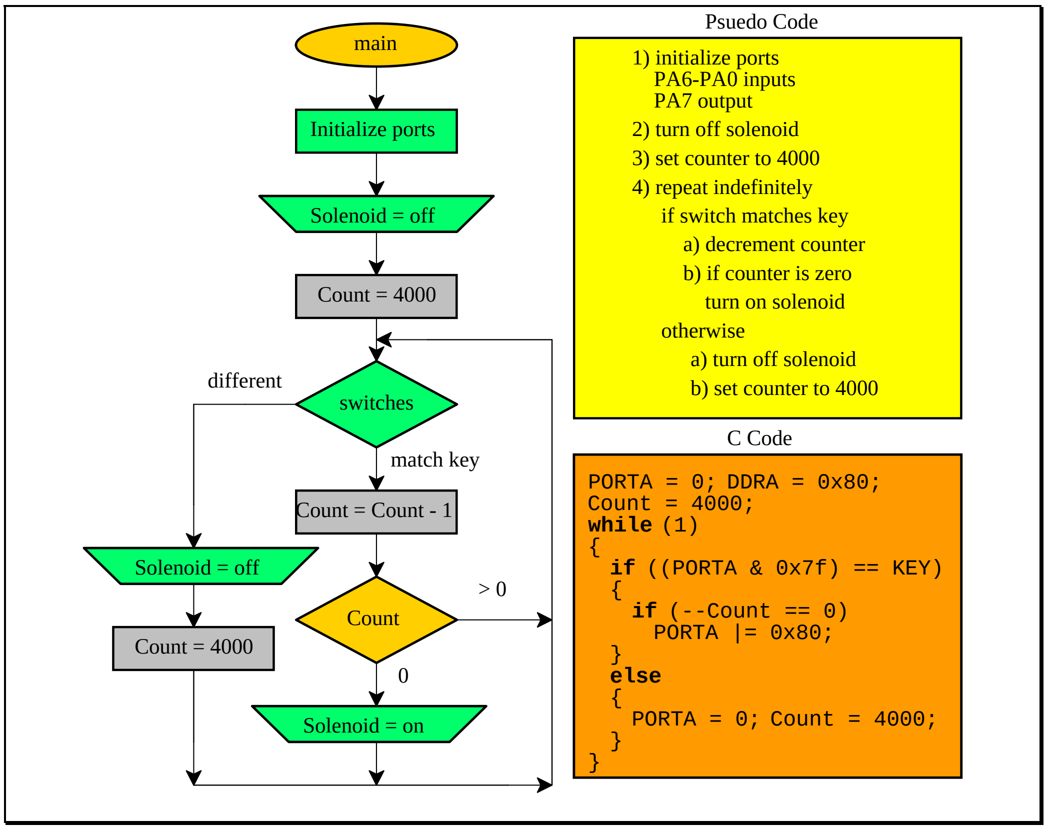
If a data structure can be defined at compile time and will remain fixed, then it can be allocated in Flash memory. In this example we will define an 8-bit fixed constant to hold the key code, which the operator needs to set to unlock the door. The compiler will place these lines with the program so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

```
const unsigned char KEY = 0x23; /* key code */
```

It is not clear at this point exactly where in Flash 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 algorithm. Experienced programmers can develop the algorithm directly in the C language. On the other hand, most of us need an abstract 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 figure shows a flowchart on the left and pseudo-code and C code on the right for our digital lock example.

A flowchart and corresponding pseudo-code and C code



Psuedo 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; DDRA = 0x80;
Count = 4000;
while (1)
{
  if ((PORTA & 0x7f) == KEY)
  {
    if (--Count == 0)
      PORTA |= 0x80;
  }
  else
  {
    PORTA = 0; Count = 4000;
  }
}
```

Figure 2.2 – Digital lock software

Normally we place the programs in Flash memory. Typically, the compiler will initialize the stack pointer to the last location of RAM. On the MC9S12A512, the stack is initialized to `0x900`. Next we write C code to implement the algorithm as illustrated in the above flowchart and pseudo-code.

4. The last stage is debugging.

2.4

Case Study 2: A Serial Port MC9S12 Program

Let's begin with a small program. This simple program is typical of the operations we perform in an embedded system. This program will read 8-bit data from parallel port C and transmit the information in serial fashion using the serial communication interface (SCI). The numbers in the first column are not part of the software, but added to simplify our discussion.

Sample serial port program for the MC9S12

```
1   /* Translates parallel input data to serial outputs */
2
3   #define PORTC *(unsigned char volatile *)(0x1003)
4   #define DDRC *(unsigned char volatile *)(0x1007)
5   #define BAUD *(unsigned char volatile *)(0x102B)
6   #define SCCR2 *(unsigned char volatile *)(0x102D)
7   #define SCSR *(unsigned char volatile *)(0x102E)
8   #define SCDR *(unsigned char volatile *)(0x102F)
9   #define TDRE 0x80
10
11  void SCI_Init(void)
12  {
13    /* 9600 baud, 16 MHz Xtal assumed */
14    BAUD = 0x34;
15    /* enable SCI, no interrupts */
16    SCCR2 = 0x0C;
17  }
18
19  /* Data is 8 bit value to send out serial port */
20  void SCI_Out(unsigned char Data)
21  {
22    /* Wait for TDRE to be set */
23    while ((SCSR & TDRE) == 0);
24    /* then output */
25    SCDR = Data;
26  }
27
28   void main(void)
29   {
30     unsigned char Info;
31     /* turn on SCI serial port */
32     SCI_Init();
33     /* specify Port C as input */
34     DDRC = 0x00;
35     while (1)
36     {
37       /* input 8 bits from parallel port C */
38       Info = PORTC;
39       /* output 8 bits to serial port */
40       SCI_Out(Info);
41     }
42   }
```

Listing 2.1 – Sample CodeWarrior Program

The first line of the program is a *comment* giving a brief description of its function. Lines 2 through 8 define *macros* that provide programming access to I/O ports of the MC9S12. These macros specify the format (unsigned 8 bit) and address (the Freescale microcomputers employ memory mapped I/O). The `#define` invokes the preprocessor to replace each instance of `PORTC` with `*(`**`unsigned char volatile`**` *)(0x1003)`.

Lines 11-17 define a *function* or procedure that when executed will initialize the SCI port. The assignment statement is of the form `value at address = data`. In particular line 14 (`BAUD = 0x30;`) will output a hexadecimal `$30` to I/O configuration register at location `$102B`. Similarly line 16 will output a hexadecimal `$0C` to I/O configuration register at location `$102D`. Notice that comments can be added virtually anywhere in order to clarify the software function. `SCI_Open` is an example of a function that is executed only once at the beginning of the program.

Line 9 is another `#define` that specifies the transmit data ready empty (`TDRE`) bit as bit 7. This `#define` illustrates the usage of macros that make the software more readable. Line 19 is a comment. Lines 20-26 define another function, `SCI_Out`, having an 8-bit input parameter that when executed will output the data to the SCI port. In particular line 23 will read the SCI status register at `$102E` over and over again until bit 7 (`TDRE`) is set. Once `TDRE` is set, it is safe to start another serial output transmission. This is an example of I/O polling. Line 25 copies the input parameter, `Data`, to the serial port, starting a serial transmission. Line 25 is an example of an I/O output operation.

Lines 28 through 42 define the main program. After some brief initialization this is where the software will start after a reset or after being powered up. The sequence **`unsigned char`** `Info` in line 30 will define a local variable. Notice that the size (**char** means 8-bit), type (**unsigned**) and name (`Info`) are specified. Line 32 calls the initialization function `SCI_Open`. Line 34 writes a `0` to the I/O configuration register at `$1007`, specifying all 8 bits of `PORTC` will be inputs (writing ones to a direction register specifies the bits as outputs). The sequence **`while`**`(1){ }` defines a control structure that executes forever and never finishes. In particular lines 37 to 40 are repeated over and over without end. Most software on embedded systems will run forever (or until the power is removed). Line 38 will read the input Port C and copy the voltage levels into the variable `Info`. This is an example of an I/O input operation. Each of the 8 lines that compose PORTC corresponds to one of the 8 bits of the variable `Info`. A digital logic high (a voltage above 2 V), is translated into a `1`. A digital logic low (a voltage less than 0.7 V) is translated into a `0`. Line 40 will execute the function `SCI_Out` that will transmit the 8-bit data via the SCI serial port.

With the CodeWarrior IDE, the system installs a reset vector address and will create code to initialize then jump to the main program automatically.

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 #), 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. This means we can place more than one statement on a single line, or place a single statement across multiple lines. For example the function `SCI_Open` could have been written without any line breaks

```
void SCI_Open(void){BAUD=0x30;SCCR2=0x0C;}
```

Use a programming style that is easy to read

Since we rarely make hardcopy printouts of our software, it is not necessary to minimize the number of line breaks. (2.1)

Similarly we could have added extra line breaks

```
void SCI_Open(void)
{
BAUD=
    0x30;
SCCR2=
   0x0C;
}
```

Just because C allows such syntax, it does not mean it is desirable. 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.

A *token* in C can be a user defined name (e.g., the variable `Info` and function `SCI_Open`) 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 and operations. 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 marks separate tokens

Punctuation	**Name**	**Meaning**
`;`	semicolon	End of statement
`:`	colon	Defines a label
`,`	comma	Separates elements of a list
`( )`	parentheses	Start and end of a parameter list
`{ }`	braces	Start and stop of a compound statement
`[ ]`	brackets	Start and stop of a array index
`" "`	quotation marks	Start and stop of a string
`' '`	apostrophes	Start and stop of a character constant

Table 2.1 – Special characters can be punctuation marks

Special characters can be operators

The next table shows the single character operators.

Operation	Name	Meaning
=	equals	Assignment statement
@	at	Address of
?	question mark	Selection
<	less than	Less than
>	greater than	Greater than
!	exclamation mark	Logical not (true to false, false to true)
~	tilde	1's complement
+	plus	Addition
-	minus	Subtraction
*	asterisk	Multiplication or pointer dereference
/	back slash	division
%	percent	Modulo, division remainder
|	pipe	Bitwise OR
&	ampersand	Bitwise AND, or address of
^	hat	Bitwise XOR
.	period	Used to access parts of a structure

Table 2.2 – Special characters can be operators

The next table shows the operators formed with multiple characters.

Multiple special characters can also be operators

Operation	**Name**	**Meaning**
`==`	is equal to	Equal to comparison
`<=`	less than or equal to	Less than or equal to
`>=`	greater than or equal to	Greater than or equal to
`!=`	not equal to	Not equal to
`<<`	shift left	Shift left
`>>`	shift right	Shift right
`++`	plus plus	Increment
`--`	minus minus	Decrement
`&&`	logical and	Boolean AND
`||`	logical or	Boolean OR
`+=`	plusequal	Add value to
`-=`	minusequal	Subtract value to
`*=`	Asterisk equals	Multiply value to
`/=`	back slash equals	Divide value to
`|=`	pipe equals	Bitwise OR value to
`&=`	ampersand equals	Bitwise AND value to
`^=`	hat equals	Bitwise XOR value to
`<<=`	shift left equals	Shift value left
`>>=`	shift right equals	Shift value right
`%=`	percent equals	Modulo divide value to
`->`	Arrow	Pointer to a part of a structure

Table 2.3 – Multiple special characters also can be operators

2.10

The following section illustrates some of the common operators. We begin with the assignment operator.

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

Listing 2.2 – Simple program illustrating C arithmetic operators

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.

Next we will introduce the arithmetic operations addition, subtraction, multiplication and division. The standard arithmetic precedencies apply.

Arithmetic operators

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

Listing 2.3 – Simple program illustrating C arithmetic operators

Next we will introduce a simple conditional control structure.

The C **if-else** control structure

```
#define PORTB *(unsigned char volatile *)(0x1004)
#define PORTE *(unsigned char volatile *)(0x100A)

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

Listing 2.4 – The C `if-else` control structure

PORTB is an output port, and PORTE is an input port on the MC9S12. 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.

Like the **if** statement, the **while** statement has a conditional test (i.e., returns a TRUE/FALSE).

The C **while** control structure

```
#define PORTA *(unsigned char volatile *)(0x0000)

void Example(void)
{
  unsigned char counter;
   /* loop until counter equals 200 */
  counter = 0;
  while (counter != 200)
  {
    /* toggle PORTA bit 3 output */
    PORTA ^= 0x08;
    /* increment counter */
    counter++;
  }
}
```

Listing 2.5 – the C `while` control structure

PORTA bit 3 is another output pin on the MC9S12. The statement immediately following the **while** will be executed over and over until the conditional test becomes FALSE.

The **for** control structure has three control expressions and a body.

The C **for** loop control structure

```
#define PORTA *(unsigned char volatile *)(0x0000)

void Example(void)
{
  unsigned char counter;
  /* loop until counter equals 200 */
  for (counter = 0; counter < 200; counter++)
  {
    /* toggle PORTA bit 3 output */
    PORTA ^= 0x08;
  }
}
```

Listing 2.6 – the C for loop control structure

The *initializer expression*, `counter = 0`, is executed once at the beginning.

The *loop test expression*, `counter < 200`, is evaluated at the beginning of each iteration through the loop, and if it is `FALSE` then the loop terminates.

Then the body, `PORTA ^= 0x08;`, is executed.

Finally, the *counting expression*, `counter++`, is evaluated at the end of each loop iteration and is usually responsible for altering the loop variable.

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.

Precedence of operators in C

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

In the preceding 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;`.

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 from 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 and associativity determine the order of operation

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 2.4 – Precedence and associativity determine the order of operation

When confused about precedence (and aren't we all?) add parentheses to clarify the expression. (2.2)

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. Lines 1 and 19 in Listing 2.1 are examples of this type of comment. 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 above each line in Listing 2.1 are examples of the second type. We place comments on separate lines so that the implementation is separate from the explanation.

Comments in the C language

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 `BAUD = 0x30;`

```
BAUD /*specifies transmission rate*/=0x30/*9600 bits/sec*/;
```

Some compilers do allow for 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:

Comments in the C++ language

```
SCI_Open(); /* turn on SCI serial port */
SCI_Open(); // turn on SCI serial port
```

We will assume (for the sake of clarity) that C++ comments are allowed in this document from now on!

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

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

Some 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 call to `SCI_Open` will result in a compiler error.

```
void main(void)
{
  unsigned char Info;
/*
  /* turn on SCI serial port */
  SCI_Open();
*/
  /* specify Port C as input */
  DDRC = 0x00;
  while (1)
  {
    // input 8 bits from parallel port C
    Info = PORTC;
    // output 8 bits to serial port
    SCI_Out(Info);
  }
}
```

The *conditional compilation* feature of a compiler 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. 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 MC9S12 I/O port names.

Preprocessor directives are processed first by the compiler

```
#include <mc9s12a256.h>
```

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 CodeWarrior include:

Global declaration objects – data or functions

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

CodeWarrior supports 32-bit long integers and floating point. We will focus on 8- and 16-bit objects. The object code generated with the compiler is often more efficient using 16-bit parameters rather than 8-bit ones.

Declarations and Definitions

Declarations specify an object – definitions define what they are (for data) or what they do (for functions)

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 `SCI_Out` could be written as

```
void SCI_Out(unsigned char);
```

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.) Notice that the function SCI_Out was defined before it was used in Listing 2.1. The following alternative approach first declares the functions, uses them, and lastly defines the functions:

```
// Translates parallel input data to serial outputs

#define PORTC *(unsigned char volatile *)(0x1003)
#define DDRC *(unsigned char volatile *)(0x1007)
#define BAUD *(unsigned char volatile *)(0x102B)
#define SCCR2 *(unsigned char volatile *)(0x102D)
#define SCSR *(unsigned char volatile *)(0x102E)
#define SCDR *(unsigned char volatile *)(0x102F)

void SCI_Open(void);                                    Function
void SCI_Out(unsigned char);                            declarations

void main(void)
{
  unsigned char Info;
  // turn on SCI serial port
  SCI_Open();
  // specify Port C as input
  DDRC = 0x00;
  while (1)
  {
    // input 8 bits from parallel port C
    Info = PORTC;
    // output 8 bits to serial port
    SCI_Out(Info);
  }
}

void SCI_Open(void)                                     Function
{                                                       definitions
  // 9600 baud
  BAUD = 0x30;
  // enable SCI, no interrupts
  SCCR2 = 0x0C;
}

// Data is 8 bit value to send out serial port
#define TDRE 0x80
void SCI_Out(unsigned char Data)
{
  // Wait for TDRE to be set
  while ((SCSR & TDRE) == 0);
  // then output
  SCDR = Data;
}
```

Listing 2.7 – Alternate C program

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 that are defined elsewhere are preceded by the keyword **`extern`**. Thus

```
short RunFlag;
```

defines a 16-bit signed integer called `RunFlag`, whereas

```
extern short RunFlag;
```

only declares `RunFlag` to exist in another, separately compiled, module. In some compilers we will use external function declarations when we create the reset/interrupt vector table. Thus the line

```
extern void TOFhandler(void);
```

Use **`extern`** to specify that an object is defined elsewhere

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 some compilers pass the first input parameter in Register D and the remaining parameters are passed on the stack. A function can have 0 or 1 output parameter. The code generated by some compilers pass the return parameter in Register D (8-bit return parameters are promoted to 16-bits.) The add function below 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.

Functions use parameters to receive input values, and sometimes return a single value

```
1  short add(short x, short y)
2  {
3    short z;
4    z = x + y;
5    if ((x > 0) && (y > 0) && (z < 0))
6      z=32767;
7    if ((x < 0) && (y < 0) && (z > 0))
8      z=-32768;
9    return(z);
10 }
11
12 void main(void)
13 {
14   short a, b;
15   a = add(2000, 2000);
16   b = 0
17   while (1)
18   {
19     b = add(b, 1);
20   }
```

Listing 2.8 – 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:

```
12   void main(void)
13   {
14     short a, b;
15     a = add(2000, 2000);            // call to add
1    short add(short x, short y)
2    {
3      short z;
4      z = x + y;                      // z = 4000
5      if (( x > 0) && (y > 0) && (z < 0))
6        z = 32767;
7      if ((x < 0) && (y < 0) && (z > 0))
8        z = -32768;
9      return(z);
10   }             // return 4000 from call
16     b = 0
17     while (1)
18     {
19       b = add(b, 1);                // call to add
1    short add(short x, short y)
2    {
3      short z;
4      z = x + y;                      // z = 4000
5      if (( x > 0) && (y > 0) && (z < 0))
6        z = 32767;
7      if ((x < 0) && (y < 0) && (z > 0))
8        z = -32768;
9      return(z);
10   }             // return 1 from call
20     }
17     while (1)
```

```
18      {
19        b = add(b, 1);                // call to add
1     short add(short x, short y)
2     {
3       short z;
4       z = x + y;                      // z = 4000
5       if (( x > 0) && (y > 0) && (z < 0))
6         z = 32767;
7       if ((x < 0) && (y < 0) && (z > 0))
8         z = -32768;
9       return(z);
10    }             // return 1 from call
20      }
```

Notice that the return from the first call goes to line 16, while all the other returns go to line 20. The execution sequence repeats lines 17, 18, 19, 1-10, 20 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 definition has two parts – a declarator and a body

A function definition 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. The following four statements are equivalent:

```
void SCI_Open(void){BAUD=0x30;SCCR2=0x0C;}
SCI_Open(void){BAUD=0x30;SCCR2=0x0C;}
void SCI_Open(){BAUD=0x30;SCCR2=0x0C;}
SCI_Open(){BAUD=0x30;SCCR2=0x0C;}
```

The **`void`** should be included as the return parameter if there is none, because it is a positive statement that the function does not return a parameter. When there are no arguments, a **`void`** should be specified to make a positive statement that the function does not require parameters.

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.

The program created by the CodeWarrior compiler actually begins execution at a place called `Init()`. 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 `Start12.c`. 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.

The C compiler always calls an `Init()` function before calling `main()`

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.

A compound statement is enclosed by `{}`

Thus, it is legal to write

Compound statements may be nested

```
// 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 2.9 – 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. We will refer to variables in the current 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.

Global variables are used for data sharing between modules

In the following example, we wish to maintain a counter of the number of times `SCI_Out` is called. 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 CodeWarrior) will automatically initialize globals to zero at startup.

Global variables are initialized by the `Init()` function

```
// number of characters transmitted
unsigned short Count;

void SCI_Open(void)
{
  // initialize global counter
  Count = 0;
  // 9600 baud
  BAUD = 0x30;
  // enable SCI, no interrupts
  SCCR2 = 0x0C;
}

#define TDRE 0x80
void SCI_Out(unsigned char Data)
{
  // Incremented each time
  Count = Count + 1;
  // Wait for TDRE to be set
  while ((SCSR & TDRE) == 0);
  // then output
  SCDR = Data;
}
```

Listing 2.10 – A global variable contains permanent information

Although the following two examples are equivalent, the second case is preferable 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.

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

Listing 2.11 – A global variable initialized at run-time by the compiler

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

Listing 2.12 – 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.

Local variables are allocated on the stack and contain temporary information

```
if (x > y)
{
  // create a temporary variable
  short z;
  // swap x and y
  z = x; x = y; y = z;
  // 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.

Local variables are local to their block and supersede the names of variables at higher levels

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 normally be used, as only small pieces of software are changed at a time. The CodeWarrior IDE supports the automatic linking of multiple source files once they are added to a project. 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.

Software usually consists of many source files

```
// ****file HC12.h ************
#define PORTC *(unsigned char volatile *)(0x1003)
#define DDRC *(unsigned char volatile *)(0x1007)
#define BAUD *(unsigned char volatile *)(0x102B)
#define SCCR2 *(unsigned char volatile *)(0x102D)
#define SCSR *(unsigned char volatile *)(0x102E)
#define SCDR *(unsigned char volatile *)(0x102F)
```

Listing 2.13 – Header file for MC9S12 I/O ports

```
// ****file SCI12.h ************
void SCI_Open(void);
void SCI_Out(unsigned char);
```

Listing 2.14 – Header file for the SCI interface

```c
// ****file SCI12.c ************
void SCI_Open(void)
{
  // 9600 baud
  BAUD = 0x30;

  // enable SCI, no interrupts
  SCCR2 = 0x0C;
}

// Data is an 8-bit value to send out the serial port
#define TDRE 0x80
void SCI_Out(unsigned char const Data)
{
  // Wait for TDRE to be set
  while ((SCSR & TDRE) == 0);

  // then output
  SCDR = Data;
}
```

Listing 2.15 – Implementation file for the SCI interface

```c
// ****file my.c ************
// Translates parallel input data to serial outputs
#include "HC12.h"
#include "SCI12.h"

void main(void)
{
  unsigned char Info;

  // turn on SCI serial port
  SCI_Open();

  // specify Port C as input
  DDRC = 0x00;
  while (1)
  {
    // input 8 bits from parallel port C
    Info = PORTC;

    // output 8 bits to serial port
    SCI_Out(Info);
  }
}
```

Listing 2.16 – Main program file for this system

This division of functions across multiple source files is clearly a matter of style.

If the software is easy to understand, debug and change, then it is written with good style.	(2.3)

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

Software style principles

- Define the I/O ports in a `HC12.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

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 SCI output function is needed in another application, then it would be a simple matter to reuse the SCI12.h and SCI12.c files. The next advantage is clarity. Compare the main program in Listing 2.16 with the entire software system in Listing 2.1. Since 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 9600 bits/sec serial port is replaced with a high-speed Universal Serial Bus (USB). For this kind of upgrade we implement the USB functions then replace the `SCI12.c` file with the new version. If we plan appropriately, we should be able to make this upgrade without changes to the files `SCI12.h` and `my.c`.

Tokens

This section defines the basic building blocks of a C program. Understanding the concepts in this section 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.

The C language is written using tokens separated by whitespace

To understand the syntax of a C program, we divide it into *tokens* separated by *white spaces* and *punctuation*. Remember that white space includes the space, tab, carriage return and line feed. 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. Finally, it creates object code that performs the intended operations. Consider the following example:

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

Listing 2.17 – Example of a function call

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

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

Since tokens are the building blocks of programs, we begin our revision of the 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 codes. 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=$09), the carriage return (13=$0D), the line feed (10=$0A), and space (32=$20).

ASCII character codes

		Bits 4 to 6							
		0	1	2	3	4	5	6	7
Bits 0 to 3	0	NUL	DLE	SP	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	“	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	’	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

Table 2.5 – ASCII Character Codes

The first 32 (values 0 to 31 or $00 to $1F) and the last one (127=$7F) are classified as *control characters*. Codes 32 to 126 (or $20 to $7E) include the "normal" characters. Normal characters are divided into

The ASCII codes are divided into control characters and normal characters

- the space character (32=$20),
- the numeric digits 0 to 9 (48 to 57 or $30 to $39),
- the uppercase alphabet A to Z (65 to 90 or $41 to $5A),
- the lowercase alphabet a to z (97 to122 or $61 to $7A), and
- the special characters (all the rest).

Literals

Literals are constants of a particular data type

Numeric literals consist of an uninterrupted sequence of digits delimited by white spaces or special characters (operators or punctuation). Although CodeWarrior does support floating-point numbers, but this document will not cover them. The use of floating-point numbers requires a substantial about of program memory and execution time – therefore most applications should be implemented using integer maths. Consequently the period will not appear in numbers as described in this document.

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.

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.

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:

Keywords are predefined tokens

Keyword	Meaning
`asm`	Insert assembly code.
`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 a global parameter as a constant in Flash, and defines a local parameter as a 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 a variable as double precision floating point.
`else`	Alternative part of a conditional.
`extern`	Defined in another module.
`float`	Specifies a 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`	16-bit integer (same as short on the MC9S12). 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 2.6 – Keywords have predefined meanings

C is case sensitive and all keywords are lowercase

Notice that all of the keywords in C are lowercase. Notice also that as a matter of style, a mixture of upper and lowercase are used for variable names, 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

Names define variables, functions and macros

We use *names* to identify our variables, functions, and macros. CodeWarrior names may be up to 31 characters long. 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 lowercase or the underscore character to create self-explaining symbols, e.g.,

```
time_of_day    go_left_then_stop
TimeOfDay      GoLeftThenStop
```

Names are case sensitive

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. The I/O port names are implemented as macros in the header file HC12.h.

Every global name defined with the CodeWarrior IDE is left as-is by the compiler. However, it defines certain names for its own use, such as startup code and library files, and precedes them with an underscore. The purpose of the underscore is to avoid clashes with the user's own global names. So, as a matter of practice, we should not ordinarily use names with leading underscores. For examples of this naming convention, observe the linker map file generated by the compiler (in the `*.map` file in the Linker Map folder in the project window).

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

1. Start every variable name with its type, like Systems Hungarian notation used by the Microsoft Windows API (abandoned with .NET). For example,

- b means Boolean true/false
- n means 8-bit signed integer
- u means 8-bit unsigned integer
- m means 16-bit signed integer
- v means 16-bit unsigned integer
- l means 32-bit integer
- p means 16-bit pointer (address)
- c means 8-bit ASCII character
- sz means null terminated ASCII string

The Systems Hungarian variable naming convention

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

3. Start every global variable and function with the 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++.

A naming convention similar to C++ objects

```
/* ********** file = Bit.c *************
   Pointer implementation of a Bit_Fifo
   These routines can be used to save (Bit_Put) and recall
   (Bit_Get) binary data 1 bit at a time (a bit stream)
   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
// storage for Bit Stream
unsigned short Bit_Fifo[Bit_FifoSize];
```

2.34

```
struct Bit_Pointer
{
  // 0x8000, 0x4000,...,2,1
  unsigned short Mask;
  // Pointer to word containing bit
  unsigned short *pWord;
};

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.pWord == p2.pWord) && (p1.Mask == p2.Mask))
    return(1);  // yes
  return(0);    // no
}

void Bit_Init(void)
{
  Bit_PutPt.Mask = Bit_GetPt.Mask = 0x8000;
  Bit_PutPt.pWord = Bit_GetPt.pWord = &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.pWord) == &Bit_Fifo[Bit_FifoSize])
      // wrap
      myPutPt.pWord = &Bit_Fifo[0];
  }
  if (Bit_Same(myPutPt, Bit_GetPt))
    // Failed, Bit_Fifo was full
    return(0);
  else
  {
    if (data)
      // set bit
      (*Bit_PutPt.pWord) |= Bit_PutPt.Mask;
    else
      // clear bit
      (*Bit_PutPt.pWord) &= ~Bit_PutPt.Mask;
    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))
    // Failed, Bit_Fifo was empty
    return(0);
  else
  {
    *datapt = (*Bit_GetPt.pWord) & Bit_GetPt.Mask;
    Bit_GetPt.Mask = Bit_GetPt.Mask >> 1;
    if (Bit_GetPt.Mask == 0)
    {
      Bit_GetPt.Mask = 0x8000;
      if ((++Bit_GetPt.pWord) == &Bit_Fifo[Bit_FifoSize])
        // wrap
        Bit_GetPt.pWord = &Bit_Fifo[0];
    }
    return(1);
  }
}
```

Listing 2.18 – A naming convention that creates modularity

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. Notice that one semicolon is placed at the end of every simple statement in the following example,

The semicolon is used as a statement terminator

```
#define PORTB *(unsigned char volatile *)(0x1004)

void Step(void)
{
  PORTB = 10;
  PORTB = 9;
  PORTB = 5;
  PORTB = 6;
}
```

Listing 2.19 – Semicolons are used to separate statements

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 (`PORTC`). We assume in this example that Port C has been initialized as an input. Semicolons are also used in the `for loop` statement, as illustrated by

The semicolon is also used in the `for` loop

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

Listing 2.20 – Semicolons are used to separate fields of the `for` statement

Colons

We can define a label using the colon. Although C has a `goto` statement, its use is strongly discouraged. 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 C input reads the same value 100 times in a row. Again we assume Port C has been initialized as an input. Notice that every time the current value on Port C is different from the previous value the counter is reinitialized.

```
char Debounce(void)
{
  short Count;
  unsigned char LastData;

Start:
  Count = 0; // number of times Port C is the same
  LastData = PORTC;
Loop:
  if (++Cnt == 100) goto Done; // same thing 100 times
  if (LastData != PORTC) goto Start; // changed
  goto Loop;
Done:
  return(LastData);
}
```

Listing 2.21 – Colons are used to define labels (places we can jump to)

Colons also terminate `case` and `default` prefixes that appear in switch statements. In the following example, the next 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);
}
```

The colon is used to terminate `case` prefixes

Listing 2.22 – Colons are also used 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. For example,

Commas are used to separate items in lists

```
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 2.23 – 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. Assuming the function `OutChar` will print a single ASCII character, the following example will print the lower case alphabet:

Apostrophes are used to specify character literals

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

Listing 2.24 – Apostrophes are used to specify characters

Quotation marks

Quotation marks are used to specify string literals. For example

Quotation marks are used to specify string literals

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

Listing 2.25 – 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 spaced to the left by two spaces. 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 2.18 and the median function in Listing 2.9.

Braces are used to create compound statements

Brackets

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

Brackets enclose array dimensions and subscripts

```
short FIFO[100];
```

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

```
PutPt = FIFO;
```

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.

Parentheses enclose argument lists and control the order of expression evaluation

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

Special characters are used for expression operators

The special characters used as *expression operators* are covered in the operator section further on in this document. 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
```

It is therefore advisable to separate operators with white space.

Numbers, Characters and Strings

This section 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. For example,

The three types of literals – numeric, character and string

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

are examples of a number literal, a character literal and a string literal respectively. The following sections discuss the way data is stored in the computer as well as the C syntax for creating the literals. The CodeWarrior compiler recognizes 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, 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

Binary information is represented by 1's and 0's

Numbers are stored in 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* (DAC) 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 $\lceil x \rceil$ be the greatest integer of x. E.g., $\lceil 2.1 \rceil$ is rounded up to 3. Tables 3.1a and 3.1b illustrate various representations of precision.

Various representations of precision

Binary Bits	Bytes	Alternatives
8	1	256
10		1,024
12		4,096
16	2	65,536
20		1,048,576
24	3	16,777,216
30		1,073,741,824
32	4	4,294,967,296
n	$\lceil n/8 \rceil$	2^n

Table 2.7 – Relationships between various representations of precision

Decimal Digits	Alternatives
3	1,000
3½	2,000
3¾	4,000
4	10,000
4½	20,000
4¾	40,000
5	100,000
n	10^n

Table 2.8 – Relationships between various representations of precision

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”	$2^{10} = 1{,}024$	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 2.9 – Common abbreviations for large numbers

8-bit unsigned numbers

A byte contains 8 bits

A byte is 8 bits

b7	b6	b5	b4	b3	b2	b1	b0

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

The value of an unsigned byte

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

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	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 2.10 – Example conversions of unsigned 8-bit binary numbers

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

The basis of an unsigned byte

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

Converting a decimal number to binary using the unsigned basis

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 see 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 begin with the largest basis element (in this case 128) and see 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 to the next largest basis element, 64 and see if we need it. We do need 64 to generate 100, so bit 6 is one and we subtract 64 from 100 to get 36. We go to the next basis element, 32 and see if we need it. Again, we do need 32 to generate 36, so bit 5 is one and we perform 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 01100100_2 (which means 64 + 32 + 4).

This operation can be visualized using the table below.

Number	Basis	Need it?	Bit	Operation
100	128	no	bit7 = 0	none
100	64	yes	bit6 = 1	100 – 64 = 36
36	32	yes	bit5 = 1	36 – 32 = 4
4	16	no	bit4 = 0	none
4	8	no	bit3 = 0	none
4	4	yes	bit2 = 1	4 – 4 = 0
0	2	no	bit1 = 0	none
0	1	no	bit0 = 0	none

Table 2.11 – Example conversion from decimal to unsigned 8-bit binary

If the least significant bit is zero, then the number is even. (2.4)

If the right-most *n* bits (least significant) are zero, then the number is divisible by $2n$**.** (2.5)

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 an 8-bit unsigned value. For example

Defining an unsigned byte in C

```
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

The value of a signed byte

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

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	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 2.12 – Example conversions of signed 8-bit binary numbers

For the signed 8-bit number system the basis is

The basis of a signed byte

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

The most significant bit in a 2's complement signed number will specify the sign. (2.6)

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 cannot 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 or C 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 (instructions) for unsigned and signed operations. For example, the MC9S12 has both unsigned `mul`, and signed

`smul`, multiply instructions. So if you use the **`smul`** instruction, you are implementing signed arithmetic. The compiler will automatically choose the proper implementation.

It is always good programming practice to have a 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 and unsigned operations may be different

	Signed different from unsigned
`/ %`	division
`*`	multiplication
`>`	greater than
`<`	less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`>>`	right shift

	Signed same as unsigned
`+`	addition
`-`	subtraction
`==`	is equal to
`|`	logical OR
`&`	logical AND
`^`	logical XOR
`<<`	left shift

Table 2.13 – Operations on signed and unsigned numbers differ

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 start with the largest basis element (in this case -128) and decide if we need to include it to make -100. 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 to the next largest basis element, 64 and see if we need it. We do not need 64 to generate 28, so bit6 is zero. We go to the next basis element, 32 and see if we need it. We do not need 32 to generate 28, so bit5 is zero. Now we need the basis element 16, so we set bit4, and subtract 16 from 28 (28 – 16 = 12). Continuing along, we need basis elements 8 and 4 but not 2 and 1, so bits 3210 are 1100. Putting it together we get 10011100$_2$ (which means -128+16+8+4).

Converting a decimal number to binary using the signed basis

This operation can be visualized using the table below.

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

Table 2.14 – Example conversion from decimal to signed 8-bit binary

To make the negative of a 2's complement signed number we first complement (toggle) all the bits, then add 1. (2.7)

Converting a decimal number to binary using 2's complement

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 01100100_2. The 2's complement negate is a two step process. First, we do a logic complement (toggle all bits) to get 10011011_2. Then, add one to the result to get 10011100_2.

Converting a decimal number to binary using modulo arithmetic

A third way to convert negative numbers into binary is to first add the number to 256, then convert the unsigned result to binary using the unsigned method. For example, to find -100, we add -100 to 256 to get 156. Then we convert 156 to binary resulting in 10011100_2. This method works because in 8-bit binary maths adding 256 to a number does not change the value.

An error will occur if you use signed operations on unsigned numbers, or use unsigned operations on signed numbers. (2.8)

To improve the clarity of software, always specify the format of data (signed versus unsigned) when defining or accessing the data. (2.9)

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

Defining a signed byte in C

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

16 bit unsigned numbers

A word or double byte contains 16 bits

A word is 16 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

The value of an unsigned word

$$\begin{aligned} N = {} & 32768 \cdot b15 + 16384 \cdot b14 + 8192 \cdot b13 + 4096 \cdot b12 \\ & + 2048 \cdot b11 + 1024 \cdot b10 + 512 \cdot b9 + 256 \cdot b8 \\ & + 128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 \\ & + 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0 \end{aligned}$$

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	0x00	0	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 2.15 – Example conversions of unsigned 16-bit binary numbers

For the unsigned 16-bit number system the basis is

The basis of an unsigned word

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

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 a 16-bit unsigned value. For example

Defining an unsigned word in C

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

16-bit signed numbers

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

The value of a signed word

$$\begin{aligned} N = & -32768 \cdot b15 + 16384 \cdot b14 + 8192 \cdot b13 + 4096 \cdot b12 \\ & + 2048 \cdot b11 + 1024 \cdot b10 + 512 \cdot b9 + 256 \cdot b8 \\ & + 128 \cdot b7 + 64 \cdot b6 + 32 \cdot b5 + 16 \cdot b4 \\ & + 8 \cdot b3 + 4 \cdot b2 + 2 \cdot b1 + b0 \end{aligned}$$

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	0x00	0	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	-31742
1111 1111 1111 1111	0xffff	-32768+16384+8192+4096+2048+1024 +512+256+128+64+32+16+8+ 4+2+1	-1

Table 2.16 – Example conversions of signed 16-bit binary numbers

For the signed 16-bit number system the basis is

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

The basis of a signed word

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;
}
```

Defining a signed word in C

Typedefs for Signed and Unsigned Data Types

To avoid confusion and make the signed and unsigned data types easy to recognise, we make the following type definitions:

```
// Signed types
typedef char INT8;
typedef int INT16;
typedef long INT32;

// Unsigned types
typedef unsigned char UINT8;
typedef unsigned int UINT16;
typedef unsigned long UINT32;
```

`typedef`s for signed and unsigned data

Big- and Little-Endian

Big- and little-endian defined

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. Intel microcomputers implement the *little-endian* approach that stores the least significant part first. The PowerPC is *bi-endian*, because it 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

Example of big- and little-endian storage of a 16-bit number

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

Example of big- and little-endian storage of a 32-bit number

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 ‘`6812`’ which is `0x36383132` at locations `0x50-0x53`, then the ASCII ‘`6`’ = `0x36` comes first in both big- and little-endian schemes.

Example of big- and little-endian storage of a multiple byte structure

address	contents
`0x0050`	`0x36`
`0x0051`	`0x38`
`0x0052`	`0x31`
`0x0053`	`0x32`

Big- and Little-endian

The term "Big-Endian" comes from Jonathan Swift’s satiric novel *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.

An error will occur when data is stored in Big-Endian format by one computer and read in Little-Endian format on another. (2.10)

Boolean information

A Boolean type has logical values of true or false

A Boolean number has two states. The two values could represent the logical values of 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 then 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 CodeWarrior define:

The Boolean type as implemented in C

False to be **all zeros**, and

True to be any **nonzero value**.

Many programmers add the following macros to their code

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

or the following enumeration, which allows for strict type and range checking

```
typedef enum {FALSE, TRUE} BOOL;
```

Decimal Numbers

Decimal numbers have notation to specify the type

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 an `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 an `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.

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 65536U	16 bits	0, 2000, 2000U, 50000U
`int`	-32768 to 32767	16 bits	-1000, 0, 1000, +20000
`unsigned short`	0 to 65536U	16 bits	0, 2000, 2000U, 50000U
`short`	-32768 to 32767	16 bits	-1000, 0, 1000, +20000
`long`	-2147483648L to 2147483647L	32 bits	-1234567L, 0L, 1234567L

Table 2.17 – The range of decimal numbers

Because the MC9S12 microcomputers are most efficient for 16-bit data (and not 32-bit data), the **`unsigned int`** and **`int`** data types are 16 bits. On the other hand, on a x86-based machine, the **`unsigned int`** and **`int`** data types are 32 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.

In C, the **`int`** data type should be avoided because it is platform dependent

Type	Freescale MC9S12	Intel x86
`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 2.18 – Differences between a MC9S12 and an x86

Since the MC9S12 microcomputers do not have direct support of 32-bit numbers, the use of long data types should be minimized. On the other hand, a careful observation of the code generated yields the fact that the compilers are more efficient with 16-bit numbers than with 8-bit numbers.

Minimize the use of 32-bit numbers on a 16-bit computer

2.56

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
}
```

The MC9S12 code generated by the CodeWarrior compiler is quite efficient when dealing with 32-bit long integers

```
ldab  #97
clra
std   I
std   J
stab  K
stab  L
std   M:2
clrb
std   M
rts
```

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.

Octal numbers begin with a leading `0`

Type	Range	Precisio n	Examples
`unsigned char`	0 to 0377	8 bits	0, 010, 0123
`char`	-0200 to 0177	8 bits	-0123, 0, 010, +010
`unsigned int`	0 to 0177777U	16 bits	0, 02000, 0150000U
`int`	-077777 to 77777	16 bits	-01000, 0, 01000, +020000
`unsigned short`	0 to 0177777U	16 bits	0, 02000, 0150000U
`short`	-077777 to 77777	16 bits	-01000, 0, 01000, +020000
`long`	-017777777777L to 017777777777L	32 bits	-01234567L, 0L, 01234567L

Table 2.19 – 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.

Each octal digit maps to 3 bits

Hexadecimal Numbers

Hexadecimal numbers are base 16, and code into a 4-bit nibble

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 humans to represent binary information, because it is extremely simple to convert back and forth between binary and hexadecimal. A *nibble* is defined as 4 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 2.20 – Definition of hexadecimal representation

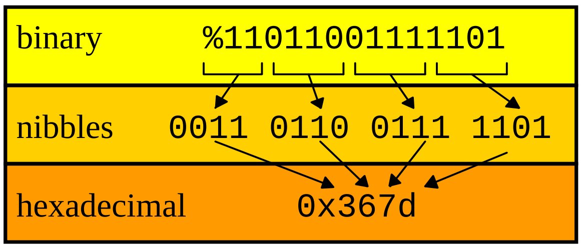
Computer programming environments use a wide variety of symbolic notations to specify the numbers in various bases. The following table illustrates various formats for numbers.

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 2.21 – Various hexadecimal formats

To convert from binary to hexadecimal we can:

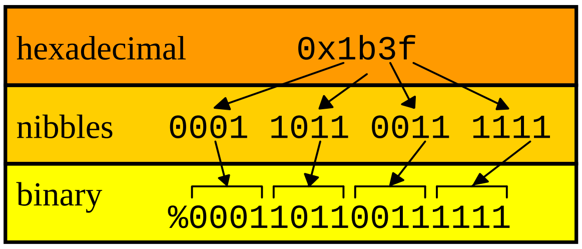
1) divide the binary number into right justified nibbles;
2) convert each nibble into its corresponding hexadecimal digit.



Converting from binary to hexadecimal

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.



Converting from hexadecimal to binary

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 or 16-bit words.

In C hexadecimal values begin with `0x`

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 int`	0x0000 to 0xffff	16 bits	0x22, 0Xabcd, 0xf0a6
`int`	-0x8000 to 0x7fff	16 bits	-0x22, 0X0, +0x70A6
`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 2.22 – 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

Character literals are surrounded by apostrophes

```
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
}
```

The MC9S12 code generated by the CodeWarrior compiler is as follows

```
ldab   #97
clra
std    I
std    J
stab   K
stab   L
std    M:2
clrb
std    M
rts
```

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.

The C char type is signed and is therefore different to unsigned ASCII literals if the sign bit is set

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., `"John"`, 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. CodeWarrior will place the strings into the text area, i.e., the string literals are considered constant and will be defined in the Flash memory 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:

Strings are really character arrays

```
char *pt;
void main(void)
{
  pt = "John"; // pointer to the string
  printf(pt);  // passes the pointer, not the data itself
}
```

Strings are surrounded by quotes

The MC9S12 code generated by the CodeWarrior compiler is as follows

```
movw   #"John", pt
ldd    pt
pshd
jsr    printf
puld
rts
```

The compiler places the string in memory and uses a pointer to it when calling `printf`. CodeWarrior pushes the parameter on the stack.

Strings in Flash memory are referenced by a pointer in RAM

Notice that the pointer, `pt`, is allocated in RAM (`.bss`) and the string is stored in Flash memory (`.text`). The assignment statement `pt="John";` copies the address, not the data. Similarly, the function `printf()` must receive the address of a string as its first (in this case, only) argument. First, the address of the string is assigned to the character pointer `pt`. Unlike other languages, the string itself is not assigned to `pt`, only its address is. After all, `pt` is a 16-bit object and, therefore, cannot hold the string itself. The same program could be written better as

```
void main(void)
{
  printf("John"); // passes the pointer, not the data itself
}
```

Notice again that the program passes a pointer to the string into `printf()`, and not the string itself. The MC9S12 code generated by the CodeWarrior compiler is as follows

```
ldd    #"John"
pshd
jsr    printf
puld
rts
```

In this case, it is tempting to think that the string itself is being passed to `printf();` but, as before, only its address is.

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.

In C, strings are null terminated (end with a 0x00 byte)

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 `"John"` sets up an array of five characters (`'J'`, `'o'`, `‘h’`, `'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)
}
```

The 6812 code generated by the CodeWarrior compiler is as follows

```
movw   #"A", pt
ldab   #65
stab   letter
rts
```

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 CodeWarrior compiler:

Escape sequences access the special ASCII codes and begin with a \

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 double quote	0x22 = 34
`\\`	ASCII back slash	0x5c = 92
`\’`	ASCII single quote	0x27 = 39

Table 2.23 – The escape sequences supported by CodeWarrior

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("\tJohn\n");
printf("\11John\12");
printf("\011John\012");
```

A newline character is represented by `\n`

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.

Backslash is also used to specify literal characters

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.

Variables and Constants

The purpose of this section 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 section will also present the C syntax and resulting assembly code generated by the CodeWarrior compiler.

The three types of literals – numeric, character and string

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 Flash memory) and is only capable of being examined. As we saw in the last section, 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 are easier to remember and 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 2.26 – Example showing a variable, a constant and some literals

The compiler options in the CodeWarrior IDE can be used to select the precision of each of the data formats. To be consistent, you should use **`short`** because on many computers **`int`** specifies a 32-bit parameter.

The concepts of precision and type (unsigned vs. signed) developed for numbers in the last section apply to variables and constants as well. In this section we will begin the discussion of variables that contain integers and characters. Even though pointers are similar in many ways to 16-bit unsigned integers, pointers will be treated in a later section. Although arrays and structures also fit the definition of a variable, they are regarded as collections of variables and will be discussed in later sections.

The distinction between global and local variables

The term *storage class* refers to the method by which an object is assigned space in memory. The CodeWarrior 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 all constants in Flash memory.

In the CodeWarrior IDE, we set the starting memory address for the static variables in the linker parameter `*.prm` file by specifying the `DEFAULT_RAM` *user segment*. The `DEFAULT_RAM` segment is just the entire RAM of the microcontroller and is used to store data. The program instructions will be placed in the `DEFAULT_ROM` user segment, which is normally a page of Flash memory reserved for instructions. The constants will be placed in the `ROM_VAR` user segment which is an area of Flash memory reserved for constants and string literals.

"bss" stands for "Block Started by Symbol", and is a leftover acronym from an early assembler written for an IBM mainframe computer in the 1950's. It is the name of the data section containing uninitialized variables

The CodeWarrior compiler places static variables in the `.bss` section, which we can view in the linker output `*.map` file in the "SECTION-ALLOCATION SECTION". It also places the program in the `.text` section, and constants in the `.rodata` (read only data) section. The Codewarrior linker automatically places sections into their correct segments (although we can specify the segment to use in our C source files with a `#pragma CODE_SEG` or `#pragma DATA_SEG` statement).

The CodeWarrior compiler uses the name of each static variable to define an assembler label. 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.

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

Listing 2.27 – Example showing a regular global variable

The MC9S12 code generated by the CodeWarrior compiler is as follows

```
main:
    ldd  #1000
    std  TheGlobal
    rts
```

Listing 2.28 – Example showing a global variable in assembly language

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. The CodeWarrior compiler will attach the C code in the `Start12.c` file to the beginning of every program. This software is executed first, before our `main()` program is started. We can see by observing the `Start12.c` file that the CodeWarrior compiler will clear all static variables to zero (`ZeroOut`) immediately after a hardware reset, and then copy all the values of initialized static variables from Flash to RAM (`CopyDown`).

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.

A static global can only be accessed within the file where it is defined

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

Listing 2.29 – Example showing a static global variable

This static global cannot be referenced outside the scope of this file.

The MC9S12 code generated by the CodeWarrior compiler is the same as a regular global. CodeWarrior limits access to the static global to functions defined in the same file.

```
main:
    ldd  #1000
    std  TheGlobal
    rts
```

Listing 2.30 – Example showing a static global in assembly language

A static local retains its value from one function call to another, and can only be accessed within the function where it is defined

A *static local* is similar to a 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`.

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

Listing 2.31 – Example showing a static local variable

Again the MC9S12 code generated by the CodeWarrior compiler is the same as a regular global. CodeWarrior limits access to the static local to the function in which it is defined.

```
main:
    ldd  #1000
    std  TheLocal
    rts
```

Listing 2.32 – Example showing a static local variable in assembly

All static variables are initialized to zero by code created by the compiler

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 CodeWarrior compiler will initialize all statics to zero (including static locals).

The following functions maintain such a count, and these counts cannot 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++;
}

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

Listing 2.33 –Two static local variables with the same name

The MC9S12 code generated by the CodeWarrior compiler is as follows

```
function1:
    ldx   TheCount
    inx
    stx   TheCount
    rts
function2:
    ldx   TheCount
    inx
    stx   TheCount
    rts
```

Listing 2.34 – Two static local variables with the same name in assembly

In each function, the address of `TheCount` will resolve to a unique address in RAM, so the disassembled code may look like

```
function1:
    ldx   0x0981
    inx
    stx   0x0981
    rts
function2:
    ldx   0x0983
    inx
    stx   0x0983
    rts
```

Listing 2.35 – Two static local variables with the same name in assembly

The CodeWarrior compiler limits the scope of the local variables to within their functions only.

Volatile

A **volatile** modifier is used to indicate that a variable can change due to external influences

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. This 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 `TOFhandler()`, 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 char Time;
void interrupt 16 TOFhandler(void)
{
  // every 16ms
  TFLG2 = 0x80;     // TOF interrupt acknowledge
  Time = Time + 1;
}

void main(void)
{
  TSCR1 |= 0x80; // TEN (enable)
  TSCR2 = 0x80;  // TOI arm, timer/1 (250ns)
  PACTL = 0x00;
  Time = 0;
  // wait for 100 counts of the 16 ms timer
  while (Time < 100);
}
```

Listing 2.36 – Code 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 MC9S12 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 PORTA *(unsigned char volatile *)(0x0000)
#define DDRA *(unsigned char volatile *)(0x0002)
void main(void)
{
  short i;
  DDRA = 0x00; // make Port A an input
  // collect 100 measurements
  for (i = 0; i < 100; i++)
  {
    data[i] = PORTA;  // collect ith measurement
  }
}
```

Listing 2.37 – Code showing 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.

Automatics

Automatic variables are created on the stack, and only exist locally

Automatic variables 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. Specifically, they are allocated on the MC9S12 stack by subtracting a value (one for characters, two for integers and four for long integers) from the stack pointer register (SP). Since automatic objects exist only within blocks, they can only be declared locally. An automatic variable can only be referenced (read or written to) 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 2.38 – 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 behaviour during execution.

Implementation of Automatic Variables

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. The CodeWarrior compiler generates code that references variables with respect to this stack frame. When the C function is entered, space is allocated by decrementing the stack pointer (the stack grows downwards in memory). This new value of SP then becomes the base for references to local variables that are declared within the function. The MC9S12 SP register points to the top data byte that has already been pushed – it is a "last-used" stack as opposed to a "next-available" stack.

Automatic variables are defined with respect to a local stack frame

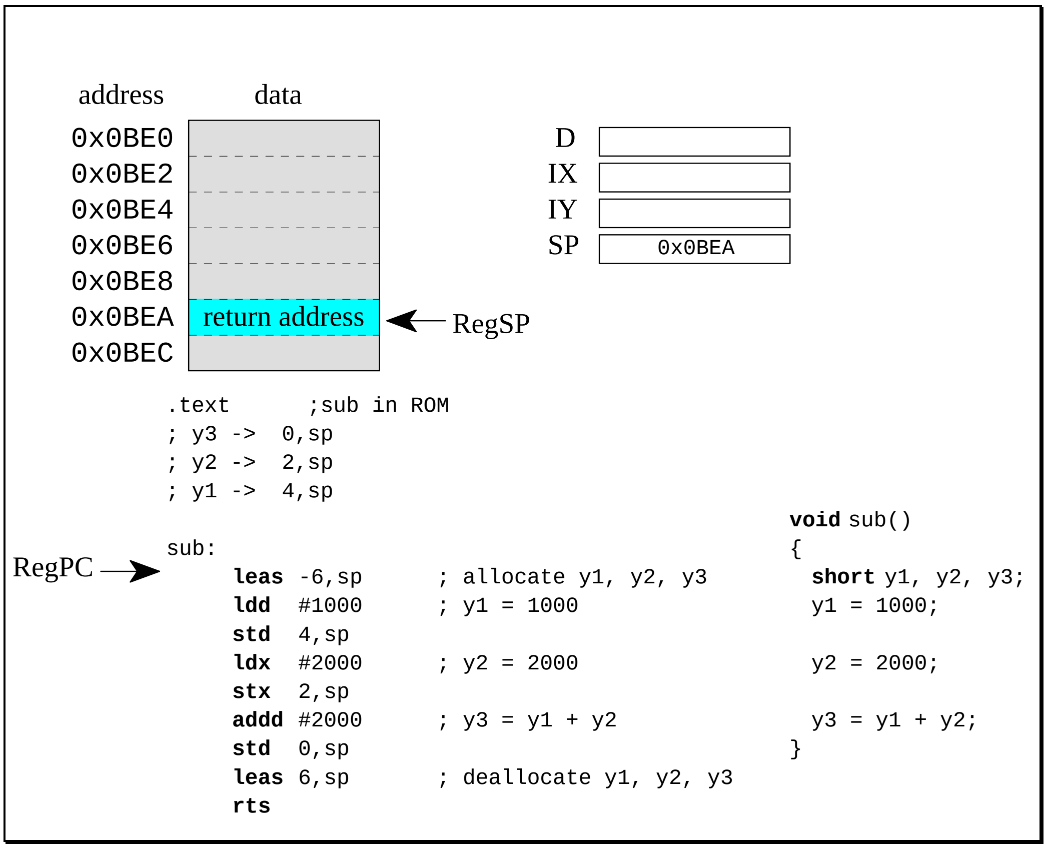
In order to understand both the machine architecture and the C compiler, we can look at the assembly code generated. For the CodeWarrior compiler, the linker/loader allocates 3 segmented memory sections: code pointed to by the PC (*.text section*); globals accessed with absolute addressing (*.data section*); and locals pointed to by the stack pointer SP. This example shows a simple C program with three local variables. Although the function doesn't do much (and will be in general be optimised out of any object code) it will serve to illustrate how local variables are created (allocation), accessed (read and write) and destroyed (deallocated.)

```c
void sub(void)
{
  short y1, y2, y3;   // 3 local variables
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

Listing 2.39 – Example showing three local variables

The disassembled output of the CodeWarrior compiler shown below has been highlighted to clarify its operation. In the MC9S12 the program counter (PC) always points to the next instruction to be executed.

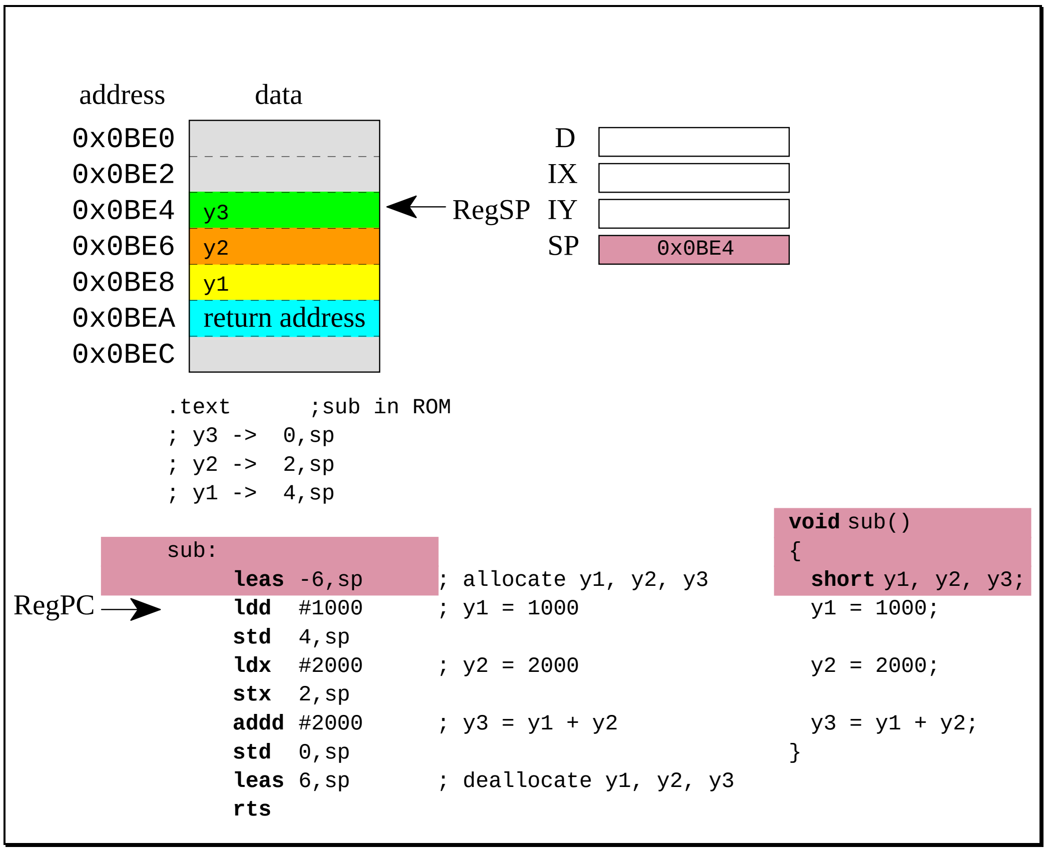
Code and MC9S12 registers and stack showing the creation and use of automatic variables



```
.text        ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

sub:
     leas  -6,sp      ; allocate y1, y2, y3
     ldd   #1000      ; y1 = 1000
     std   4,sp
     ldx   #2000      ; y2 = 2000
     stx   2,sp
     addd  #2000      ; y3 = y1 + y2
     std   0,sp
     leas  6,sp       ; deallocate y1, y2, y3
     rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;

  y2 = 2000;

  y3 = y1 + y2;
}
```

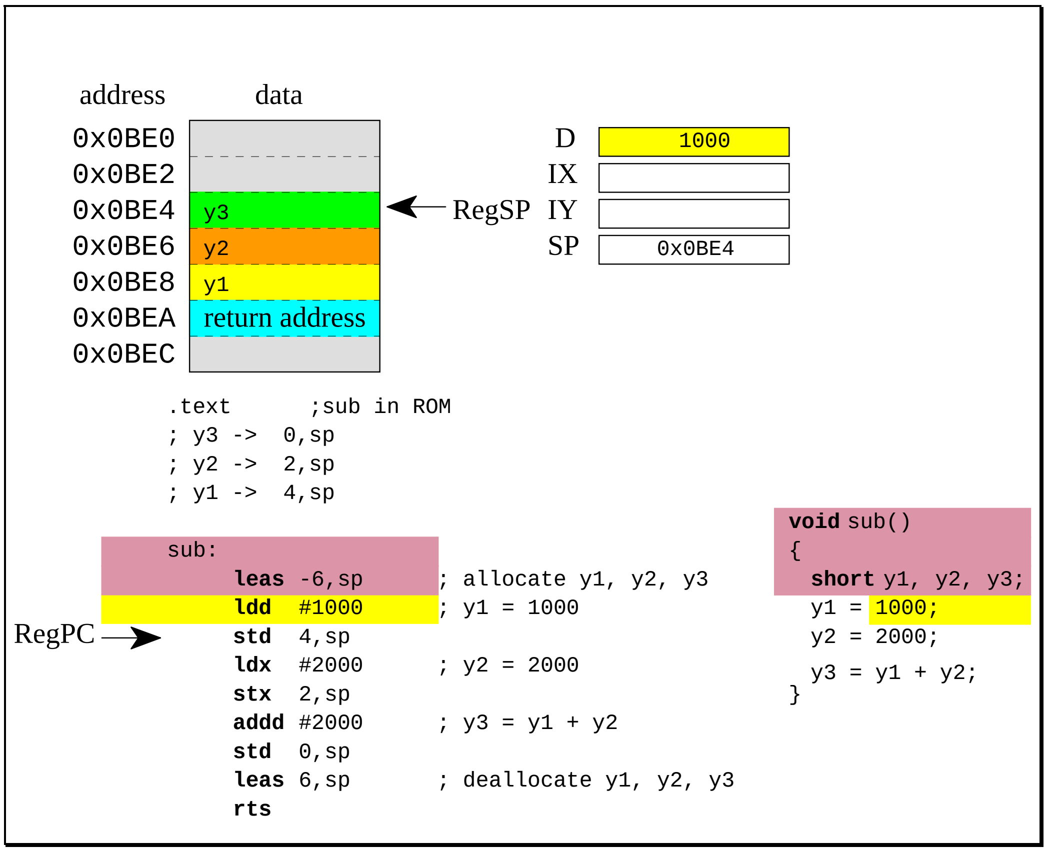
Figure 2.3 – MC9S12 implementation of three local variables – step 1



```
.text        ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

sub:
     leas  -6,sp      ; allocate y1, y2, y3
     ldd   #1000      ; y1 = 1000
     std   4,sp
     ldx   #2000      ; y2 = 2000
     stx   2,sp
     addd  #2000      ; y3 = y1 + y2
     std   0,sp
     leas  6,sp       ; deallocate y1, y2, y3
     rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;

  y2 = 2000;

  y3 = y1 + y2;
}
```

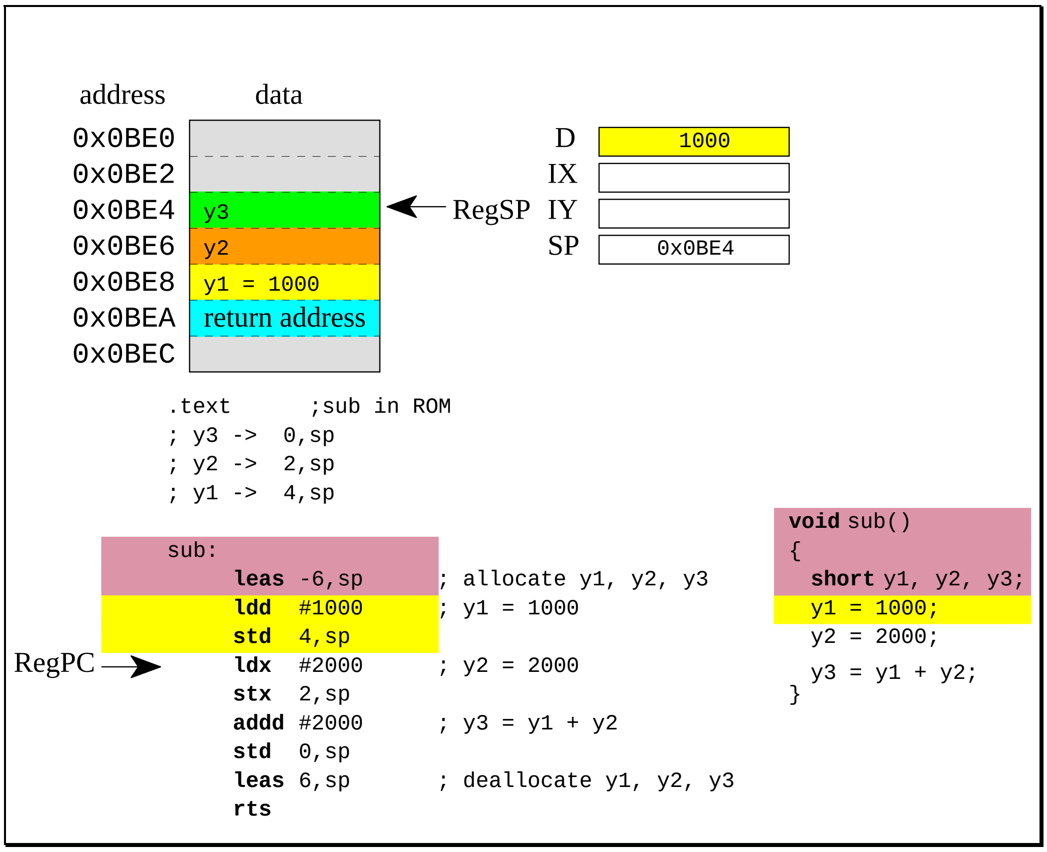
Figure 2.4 – MC9S12 implementation of three local variables – step 2



```
        .text      ;sub in ROM
        ; y3 ->  0,sp
        ; y2 ->  2,sp
        ; y1 ->  4,sp

        sub:
             leas -6,sp      ; allocate y1, y2, y3
             ldd  #1000      ; y1 = 1000
             std  4,sp
             ldx  #2000      ; y2 = 2000
             stx  2,sp
             addd #2000      ; y3 = y1 + y2
             std  0,sp
             leas 6,sp       ; deallocate y1, y2, y3
             rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

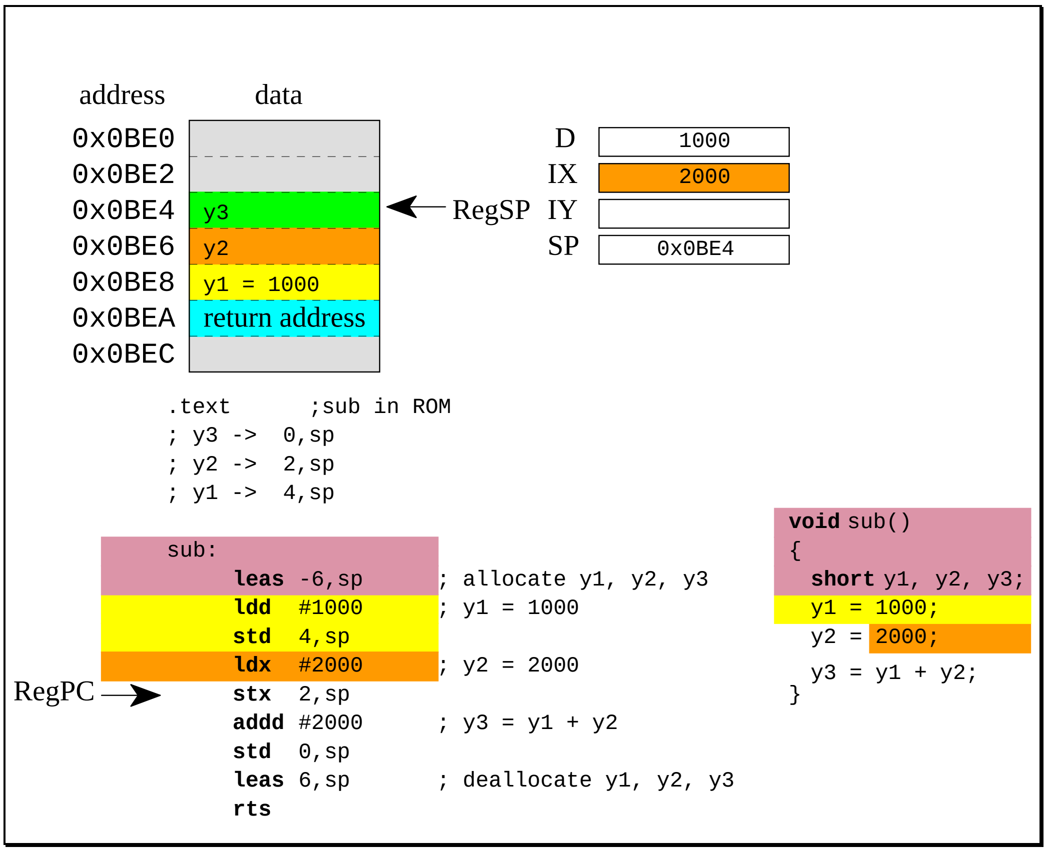
Figure 2.5 – MC9S12 implementation of three local variables – step 3



```
        .text      ;sub in ROM
        ; y3 ->  0,sp
        ; y2 ->  2,sp
        ; y1 ->  4,sp

        sub:
             leas -6,sp      ; allocate y1, y2, y3
             ldd  #1000      ; y1 = 1000
             std  4,sp
             ldx  #2000      ; y2 = 2000
             stx  2,sp
             addd #2000      ; y3 = y1 + y2
             std  0,sp
             leas 6,sp       ; deallocate y1, y2, y3
             rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

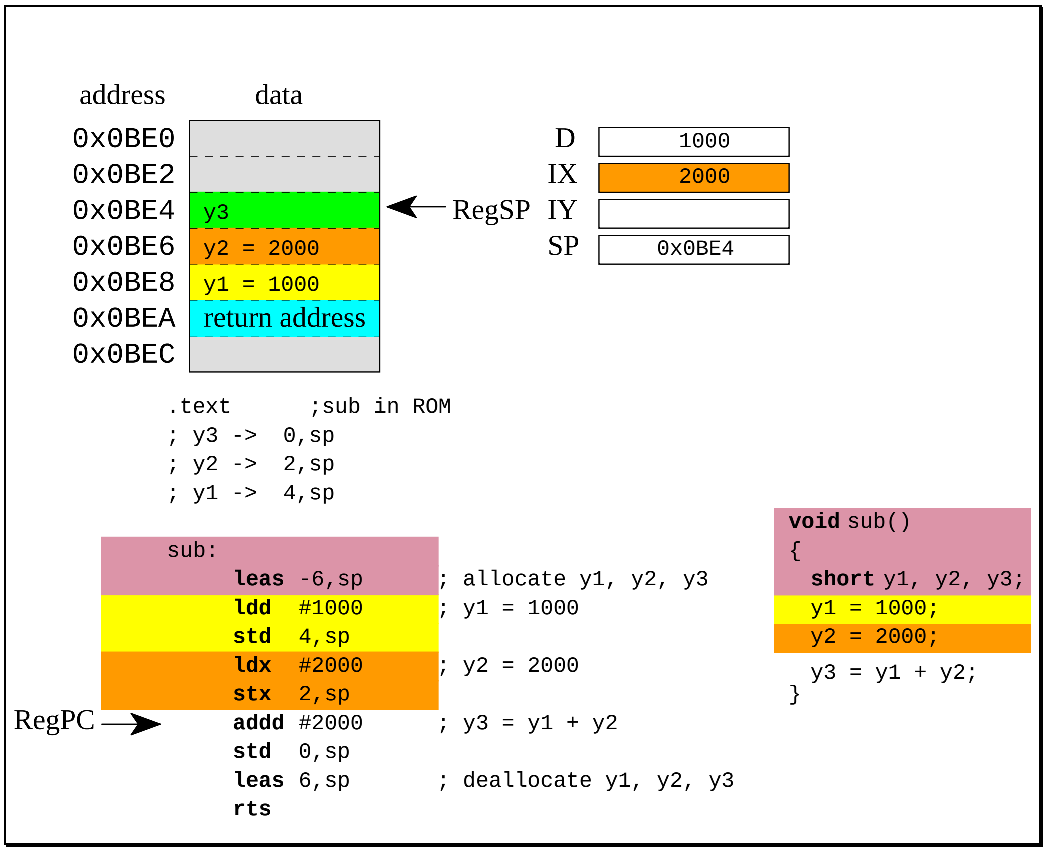
Figure 2.6 – MC9S12 implementation of three local variables – step 4



```
.text      ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

sub:
     leas -6,sp     ; allocate y1, y2, y3
     ldd  #1000     ; y1 = 1000
     std  4,sp
     ldx  #2000     ; y2 = 2000
     stx  2,sp
     addd #2000     ; y3 = y1 + y2
     std  0,sp
     leas 6,sp      ; deallocate y1, y2, y3
     rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

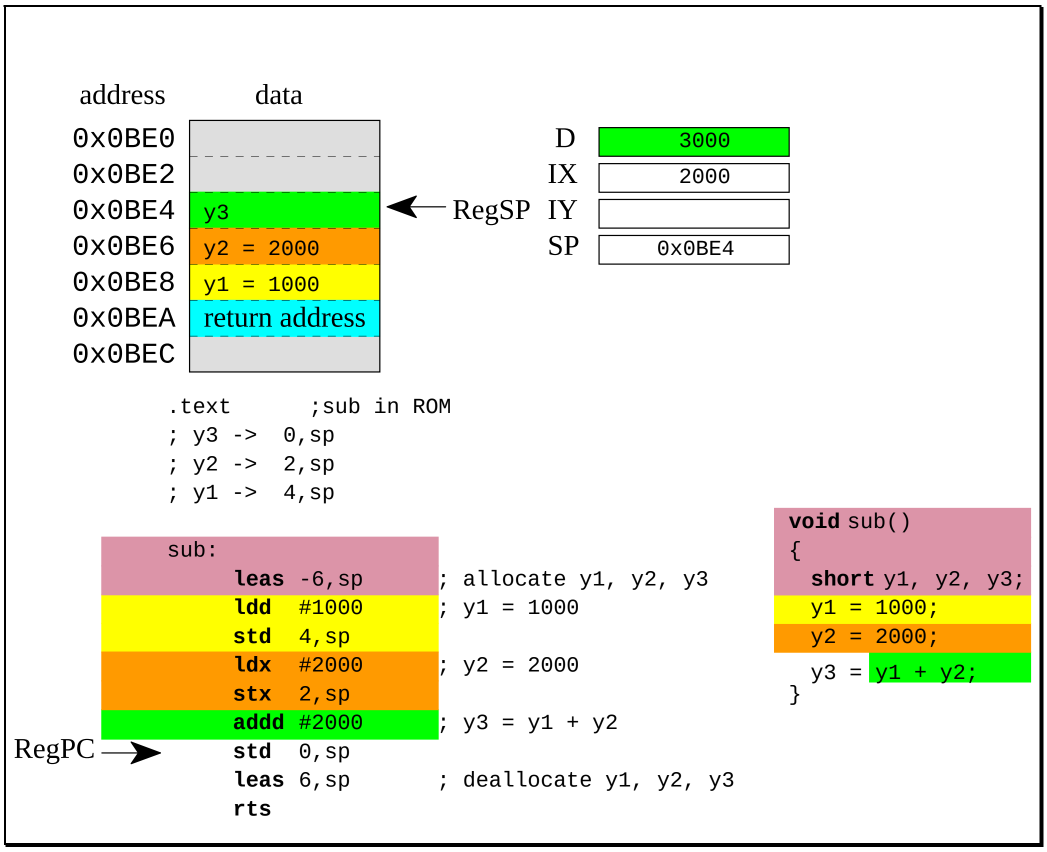
Figure 2.7 – MC9S12 implementation of three local variables – step 5



```
.text      ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

sub:
     leas -6,sp     ; allocate y1, y2, y3
     ldd  #1000     ; y1 = 1000
     std  4,sp
     ldx  #2000     ; y2 = 2000
     stx  2,sp
     addd #2000     ; y3 = y1 + y2
     std  0,sp
     leas 6,sp      ; deallocate y1, y2, y3
     rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

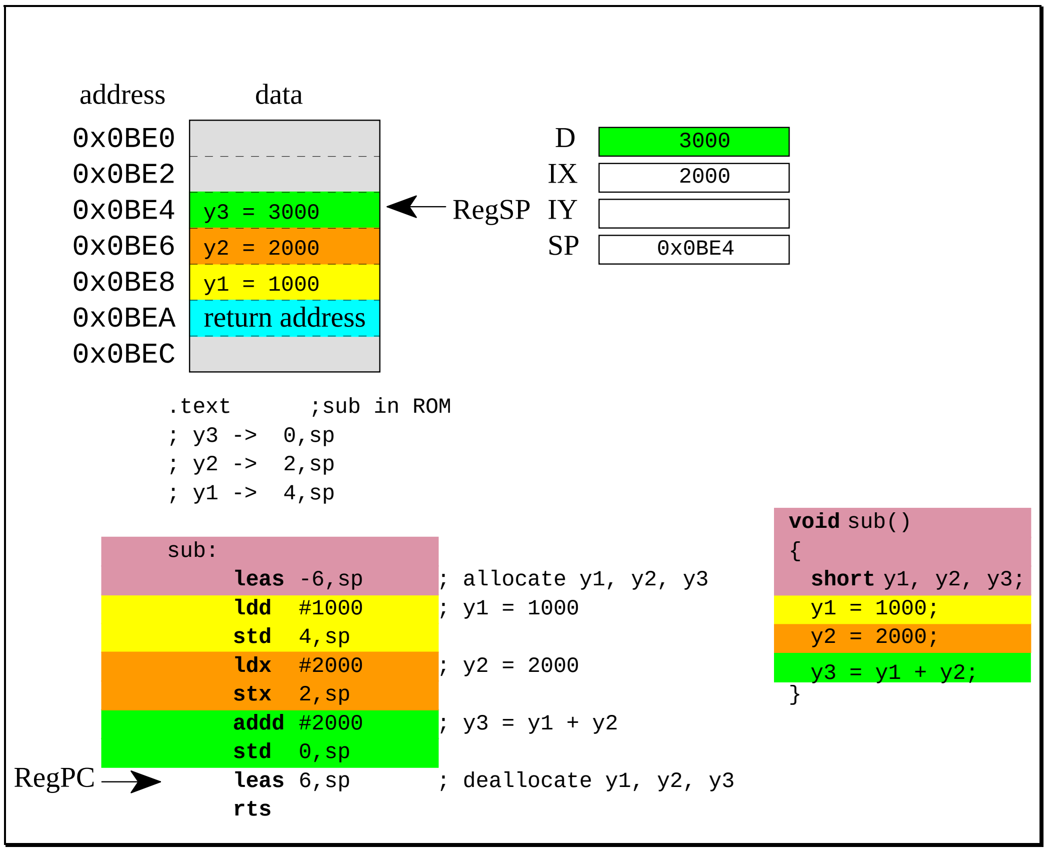
Figure 2.8 – MC9S12 implementation of three local variables – step 6



```
        .text      ;sub in ROM
        ; y3 -> 0,sp
        ; y2 -> 2,sp
        ; y1 -> 4,sp

        sub:
             leas -6,sp     ; allocate y1, y2, y3
             ldd  #1000     ; y1 = 1000
             std  4,sp
             ldx  #2000     ; y2 = 2000
             stx  2,sp
             addd #2000     ; y3 = y1 + y2
             std  0,sp
             leas 6,sp      ; deallocate y1, y2, y3
             rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

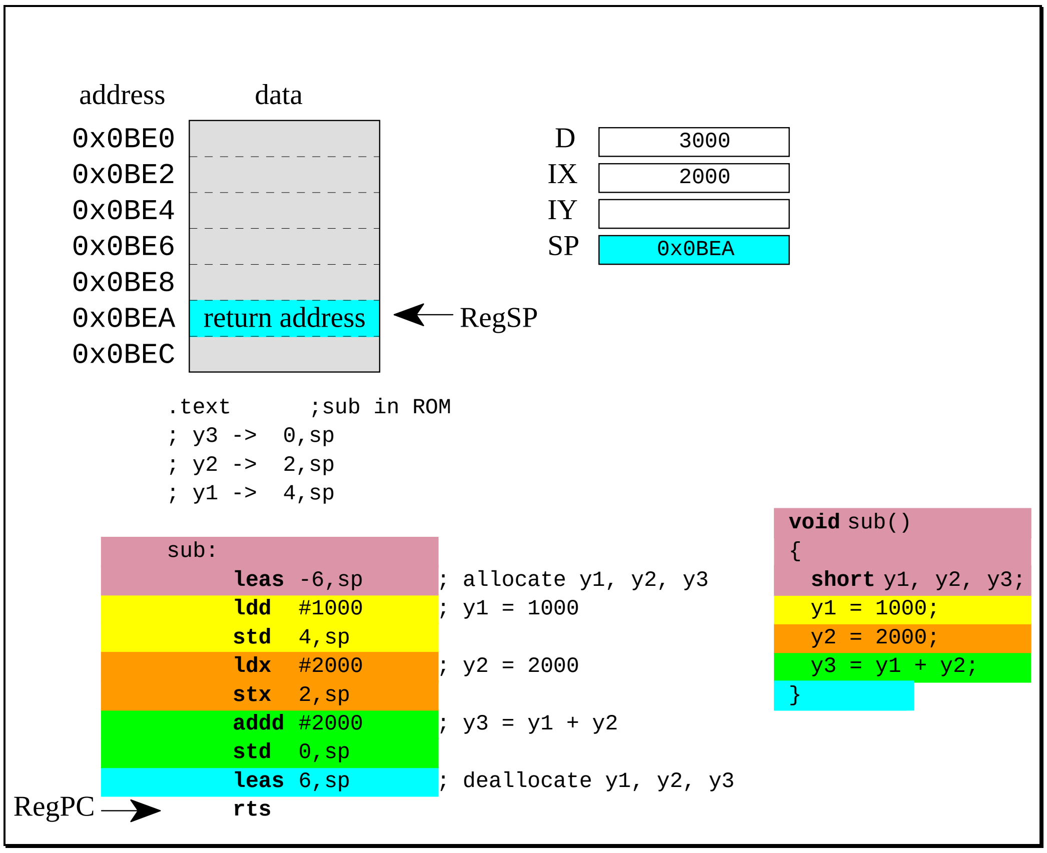
Figure 2.9 – MC9S12 implementation of three local variables – step 7



```
        .text      ;sub in ROM
        ; y3 -> 0,sp
        ; y2 -> 2,sp
        ; y1 -> 4,sp

        sub:
             leas -6,sp     ; allocate y1, y2, y3
             ldd  #1000     ; y1 = 1000
             std  4,sp
             ldx  #2000     ; y2 = 2000
             stx  2,sp
             addd #2000     ; y3 = y1 + y2
             std  0,sp
             leas 6,sp      ; deallocate y1, y2, y3
             rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

Figure 2.10 – MC9S12 implementation of three local variables – step 8

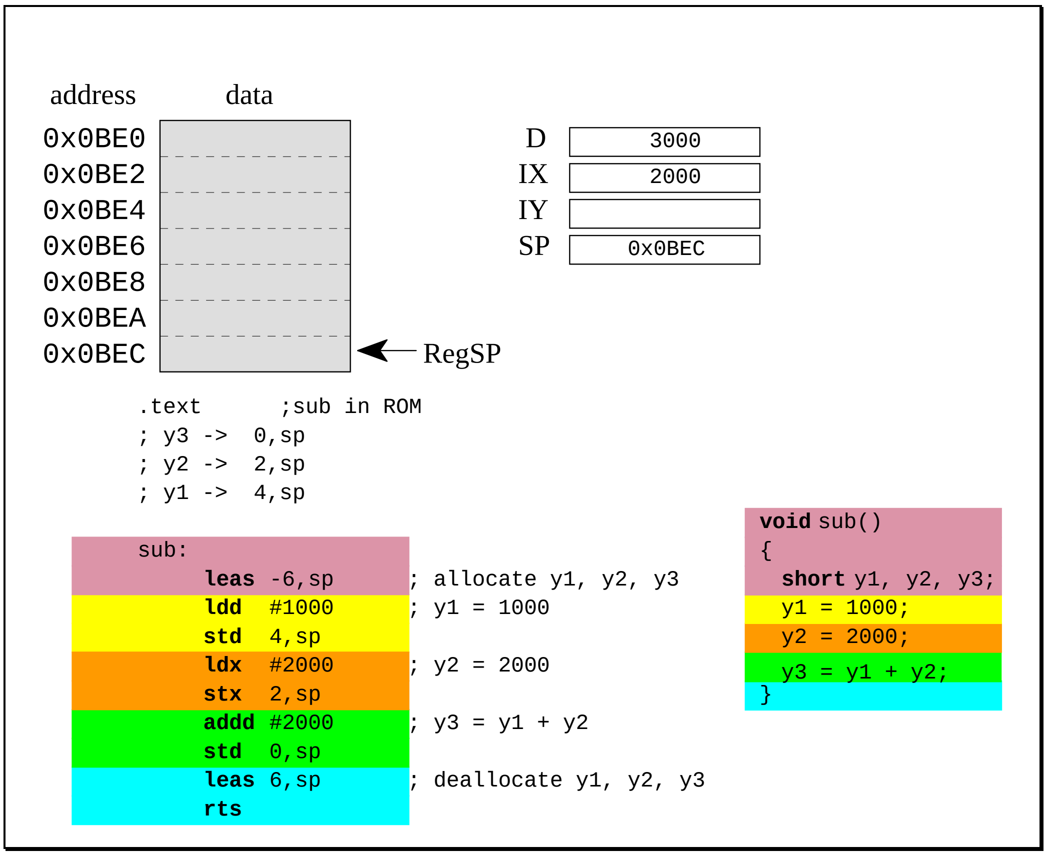


```
.text       ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

     sub:
          leas  -6,sp    ; allocate y1, y2, y3
          ldd   #1000    ; y1 = 1000
          std   4,sp
          ldx   #2000    ; y2 = 2000
          stx   2,sp
          addd  #2000    ; y3 = y1 + y2
          std   0,sp
          leas  6,sp     ; deallocate y1, y2, y3
          rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

Figure 2.11 – MC9S12 implementation of three local variables – step 9

Constant locals are defined temporarily on the stack



```
.text       ;sub in ROM
; y3 ->  0,sp
; y2 ->  2,sp
; y1 ->  4,sp

     sub:
          leas  -6,sp    ; allocate y1, y2, y3
          ldd   #1000    ; y1 = 1000
          std   4,sp
          ldx   #2000    ; y2 = 2000
          stx   2,sp
          addd  #2000    ; y3 = y1 + y2
          std   0,sp
          leas  6,sp     ; deallocate y1, y2, y3
          rts
```

```
void sub()
{
  short y1, y2, y3;
  y1 = 1000;
  y2 = 2000;
  y3 = y1 + y2;
}
```

Figure 2.12 – MC9S12 implementation of three local variables – step 10

The **leas** -6,sp instruction allocates the local variables, and thereafter they are accessed by indexing the stack pointer. Within the subroutine the local variables of other functions are not accessible. If a function is called from within another function, the new function will allocate its own local variable space on the stack, without disturbing the existing data.

Implementation of Constant Locals

A *constant local* is different to a regular local. Unlike the other locals, the constant is *not* defined temporarily on the stack. Since it cannot be changed, the assembly language code generated by the CodeWarrior compiler that references the constant local replaces the reference with the actual value.

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

Listing 2.40 – Example showing a constant local

The MC9S12 code generated by the CodeWarrior compiler is as follows (notice the reservation of space in the .bss section for the global variable)

```
    .text
sub:
    pshx
    ldd   #1000
    std   TheGlobal
    rts

    .bss
TheGlobal
```

Externals

Externals are variables defined elsewhere

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 2.41 – Example showing an external global

The assembly language the CodeWarrior compiler generates does *not* include the definition of `ExtGlobal`. The MC9S12 code generated by the CodeWarrior compiler is as follows

```
        .text
main:
        ldd    #1000
        std    ExtGlobal
        rts
```

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.

Scope refers to where a variable can be “seen”

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.

Global variables have a scope that extends from the declaration to the end of the file

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.

Local variables have a scope that is restricted to the function where they are defined

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 2.42 – Example showing the scope of local variables

Local variables can have nested scope

This program declares variables with the name x, assigns values to them, and outputs them to PORTA 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 2.43 – Example showing an illegal local variable declaration

Declarations

Every variable in C must be *declared* before it is used. Declarations force us to consider the precision (8-bit, 16-bit etc.) and format (unsigned vs. signed) of each variable.

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 previously, if the declaration is preceded by **`extern`** it only declares the type of the variable, without reserving space for it. 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.

A declaration defines the type of a variable and where it is located in memory

Table 2.24 contains examples of legitimate variable declarations. Notice that the declarations are introduced by one or two type keywords that state the data types 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.

Variables have a **`signed`** modifier by default

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;`	16-bit unsigned number	0 to +65535
`int i1, i2;`	two 16-bit signed numbers	-32768 to +32767
`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
`float f1, f2;`	two 32-bit floating-point numbers	not recommended
`double d1, d2;`	two 64-bit floating-point numbers	not recommended

Table 2.24 – Variable declarations

Storage class indicates where variables should be placed in memory

The CodeWarrior compiler allows the **`register`** modifier for automatic variables, but this is usually unnecessary as the compiler will use registers in preference to locals on the stack (for speed reasons). The keywords **`char`**, **`int`**, **`short`** and **`long`** specify the precision of the variable. The following tables shows the available storage classes and modifiers for variables.

Storage Class	**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 2.25 – 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 2.26 – Variable modifiers

A modifier is used to further classify a variable's type

In all cases **`const`** means the variable has a fixed value and cannot be changed. When modifying a global on an embedded system like the MC9S12, it also means the parameter will be allocated in Flash memory. In the following example, **`Ret`** is allocated in Flash memory. 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.

A **`const`** modifier means a variable cannot be changed

```
unsigned char const Ret = 13;
void LegalFunction(short Count)
{
  while (Count)
  {
    SCI_OutChar(Ret);
    Count--;
  }
}
```

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

```
void NotLegalFunction(const short Count)
{
  while (Count)
  {
    SCI_OutChar(13);
    Count--;  // 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 substituted by the value 5.

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

Character Variables

Characters are 8-bit quantities promoted to 16-bits

Character variables are stored as 8-bit quantities. When they are fetched from memory, they are always promoted automatically to 16-bit integers. Unsigned 8-bit values are promoted by adding 8 zeros into the most significant bits. Signed values are promoted by copying the sign bit (bit7) into the 8 most significant bits.

Do not mix signed and unsigned variables in expressions

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

Listing 2.44 – Code 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 16-bit values. In this case, however, it does not matter whether they are signed or unsigned. Obviously there is only one reasonable way to put a 16-bit quantity into an 8-bit location. When the high-order byte is 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 Flash memory. An *automatic variable* contains temporary information used only by one software module. 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 re-entrant 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 and 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:

Automatic variables contain temporary information used only by one software module, and they are interrupt proof

- dynamic allocation release allows for reuse of memory
- limited scope of access provides for data protection
- can be made re-entrant.
- 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. For example, we use globals to pass data between the main (or background) process and an interrupt (or foreground) 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, etc. The CodeWarrior compiler uses absolute addressing (direct or extended) to access the static variables.

Static variables contain information that is shared between software modules

Initialization of variables and constants

Static variables are initialized to zero

Most programming languages provide ways of specifying *initial values*; that is, the values that variables have when program execution begins. The CodeWarrior 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.

Constants must be initialized when they are declared

Specifying initial values is simple. In its declaration, we follow a variable's name with an equals 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 `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 `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.

Character arrays can be initialized with a character string

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[5] = "John";
```

declares `Name` to be a character array of five elements with the first four initialized to `'J'`, `'o'`, `‘h’` and `'n'` respectively. The fifth 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

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

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

```
char Name[7] = "John";
```

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 = "John";
```

Pointers can be initialized to point to a constant character string

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 "John". The CodeWarrior compiler accepts 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 Flash memory (normally as part of the instruction for small values). In the following example, `J` is a static constant, and `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 2.45 – Example showing the initialization of a static constant

The MC9S12 code generated by the CodeWarrior compiler is as follows

```
    .text
main:
    ldab  #96    ;8-bits
    clra         ;now 16-bits
    std   I      ;16-bits
    ldx   #97    ;16-bits
    stx   I      ;16-bits
    rts

    .bss
    I
```

Notice the use of the `#define` macro implements an operation similar to the literal `I = 97;`.

The compiler initializes a static variable by defining its initial value in Flash memory. It creates another segment called *.rodata* (in addition to the *.data* and *.text* sections). It places the initial values in the *.rodata* segment, then copies the data dynamically from *.rodata* Flash memory into *.data* RAM variables at the start of the program (before `main` is started). For example

```
short I = 95;           // 16-bit global
void main(void)
{
  ...
}
```

For the CodeWarrior compiler, code in the `Start12.c` file will copy the `95` from `.rodata` (Flash memory) into `I` in `.bss` (RAM) upon a hardware reset.

This copy is performed transparently before the main program is started.

```
        .text
    main:
        ...
        rts

        .bss
        I
        .idata
        .word 95
```

Even though the following two initializations of a global variable are technically proper, the explicit initialization of a global variable is a better style.

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

A good understanding of the assembly code generated by our compiler makes us better programmers. (2.11)

Summary of Variable Attributes

Every variable possesses a number of different attributes, as summarized in the table below:

Attribute	**Description**
Type	`char`, `int`, `unsigned int`, etc. (*also implies size, range and resolution*)
Name	The identifier used to access the variable.
Value	The data held within the variable.
Address	The location in memory where the variable resides.
Scope	That part of the source code where the variable's name is recognized.
Lifetime	A notion of when the variable is created and destroyed, and thus when it is available for use.

Table 2.27 – Attributes of variables stored in memory

Summary of Variable Lifetimes

C offers three basic types of memory allocation as summarized below:

Method	**Variable is created…**	**Variable is initialized…**	**Variable is destroyed…**
Automatic	*Each* time the program enters the *function* in which it is declared.	If specified in the declaration, initialization occurs *each* time the program enters the *block*.	*Each* time the function *returns*.
Static	*Once*: When the program is first loaded into memory.	*Once*: Just before the program starts to run.	*Once*: When the program stops.
Dynamic	By calling the library function `malloc`.	By writing executable statements that modify its content.	By calling the library function `free`.

Table 2.28 – Types of memory allocation available in C

Each was designed for a different purpose; understanding their behaviour is crucial in order to take advantage of their capability.

Expressions

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 non-traditional data types and a rich set of operators. Pointers, unsubscripted array names, and function names are allowed as operands. As Table 2.29 through to Table 2.34 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 of 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.

An expression is a combination of constants, variables array elements and function calls joined by operators

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.

Operand count refers to how many variables the operator is applied to

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. The following statement has both good style and proper calculation:

Precedence and associativity are confusing – use parentheses

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

The issues of precedence and associativity were explained in an earlier section. Precedence defines the evaluation order. For example the expression `3+4*2` will be `11` because multiplication has 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. Refer to Table 2.4 for a list of operators and their precedence and associativity.

Associativity determines order of execution for operators with the same precedence

Unary operators

Unary operators 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:

Unary operators have a single input, and a single output

```
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 not zero 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 2.29 – 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 2.30 – Unary postfix operators

Binary operators

Binary operators have two inputs, and one output

Binary arithmetic operators 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 2.31 – 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 XOR	`0x1234 ^ 0x00FF`	`0x12CB`

Table 2.32 – 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 2.33 – 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 not zero) 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).

Don't confuse bitwise logical operators with Boolean operators

```
short a, b, c, d;
void main(void)
{
  a = 0x0F0F;
  b = 0xF0F0;
  c = a & b;  // logical result c will be 0x0000
  d = a && b; // Boolean result d will be 1 (true)
}
```

Listing 2.46 – The difference between logical and Boolean operators

Binary relational operators

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

Operator	Meaning	Example	Result
`==`	is equal to	`100 == 200`	`0` (false)
`!=`	is not equal to	`100 != 200`	`1` (true)
`<`	less than	`100 < 200`	`1` (true)
`<=`	less than or equal to	`100 <= 200`	`1` (true)
`>`	greater than	`100 > 200`	`0` (false)
`>=`	greater than or equal to	`100 >= 200`	`0` (false)

Table 2.34 – Binary relational operators

Don't confuse assignment equals (=) with relational equals (==)

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 2.47 – 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. The assignment operation itself has a result, so the assignment operation can be nested.

The assignment operator

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

Listing 2.48 – Example of a nested assignment operation

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

Read / modify / write assignment operators

```
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 2.49 – 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 2.50 – Good examples of read/modify/write assignment operations

Expression Types and Explicit Casting

Declare the size of an integer type explicitly

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 2.35. Notice that for the MC9S12, the **`int`** and **`short`** types are the same. On the other hand, with the Intel Pentium, the **`int`** and **`long`** types are the same. This difference may cause confusion, when porting code from one system to another. You should 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`	`-127` to `127`	8-bits	**`char`** `sc;`
`unsigned int`	`0` to `65535U`	16-bits	**`unsigned int`** `ui;`
`int`	`-32768` to `32767`	16-bits	**`int`** `si;`
`unsigned short`	`0` to `65535U`	16-bits	**`unsigned short`** `us;`
`short`	`-32768` to `32767`	16-bits	**`short`** `ss;`
`long`	`-2147483648L` to `2147483647L`	32-bits	**`long`** `sl;`

Table 2.35 – Available number formats for the CodeWarrior compiler

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:

Names for type conversion

- 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.

Promotion of type

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 2.36 – Conversion when the range of one type fits inside another

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

Use a typecast to force a particular type conversion demotion of the 16-bit signed number into an 8-bit signed number. Unfortunately, C does not provide any simple mechanisms for error detection / correction.

```
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 2.51 – Example of converting types with the cast operator

An explicit cast is achieved by preceding the expression with parentheses surrounding the type

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. 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
// arrays containing current and previous values
short x[3], y[3];
#define OC5 0x20
void interrupt 13 TOC5handler(void)
{
  TFLG1 = OC5;      // ack OC5F
  TC5 = TC5 + 8333; // fs = 240Hz
  // shift arrays
  y[2] = y[1];
  y[1] = y[0];
  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]) / 128;
}
```

Listing 2. 52 – We can use a cast to force higher precision arithmetic

A cast is used to give a symbolic name to a microcontroller port or register

We saw previously that casting was used to assign a symbolic name to an I/O port. In particular the following `define` casts the number `0x0000` as a pointer type, which points to unsigned 8-bit data.

```
#define PORTA *(unsigned char volatile *)(0x0000)
```

Selection operator

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

The selection operator

```
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 type 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 2.53 – Example 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

Overflow and underflow can occur when performing arithmetic 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:

An overflow occurs when the result of an operation cannot fit into the finite precision of the result

- 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.

Condition code bits are used to detect an overflow or underflow

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 2.37 – 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 ±1. If we add two unsigned numbers, we start at the position of the first number and 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.

For unsigned numbers, errors occur when crossing the 0|255 boundary

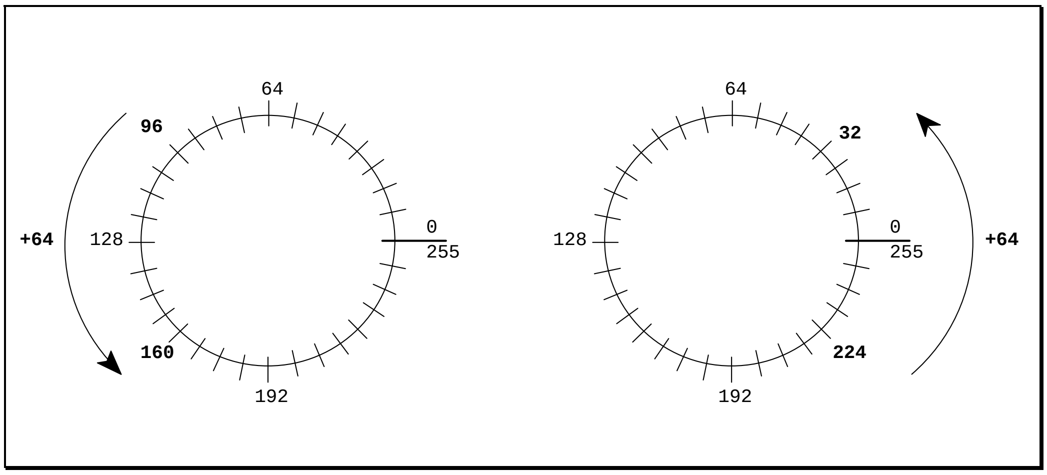


Figure 2.13 – 8-bit unsigned addition

For subtraction, we start at the position of the first number and move in a counter clockwise 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`) .

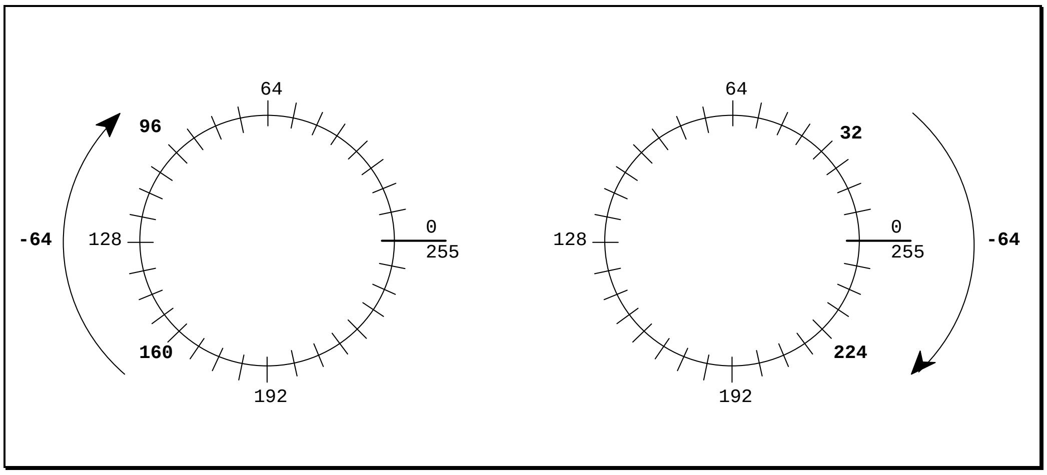


Figure 2.14 – 8-bit unsigned subtraction

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.

The carry bit, C, is set after an unsigned add or subtract when the result is incorrect. (2.12)

For signed numbers, errors occur when crossing the -128|127 boundary

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 ±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 counter clockwise 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.

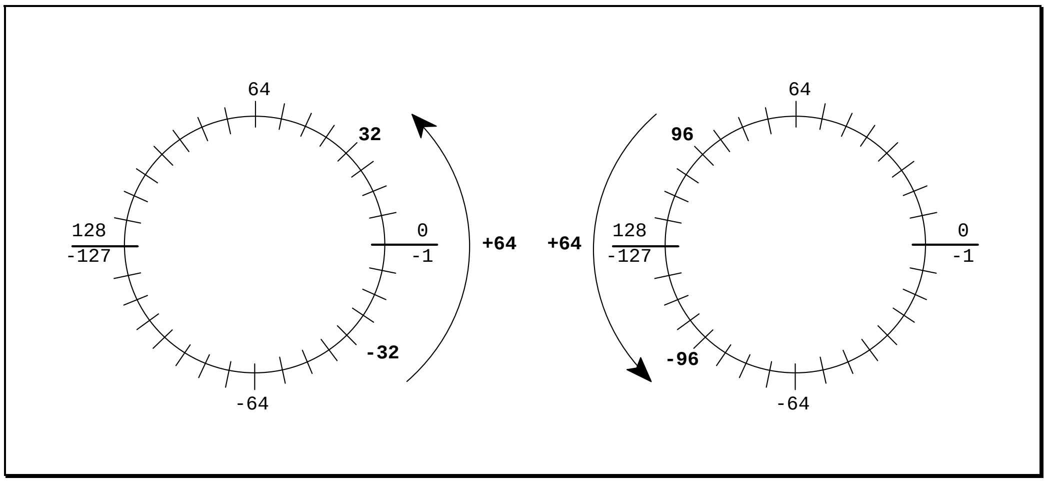


Figure 2.15 – 8-bit signed addition

For subtracting signed numbers, we again move in a counter clockwise 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`).

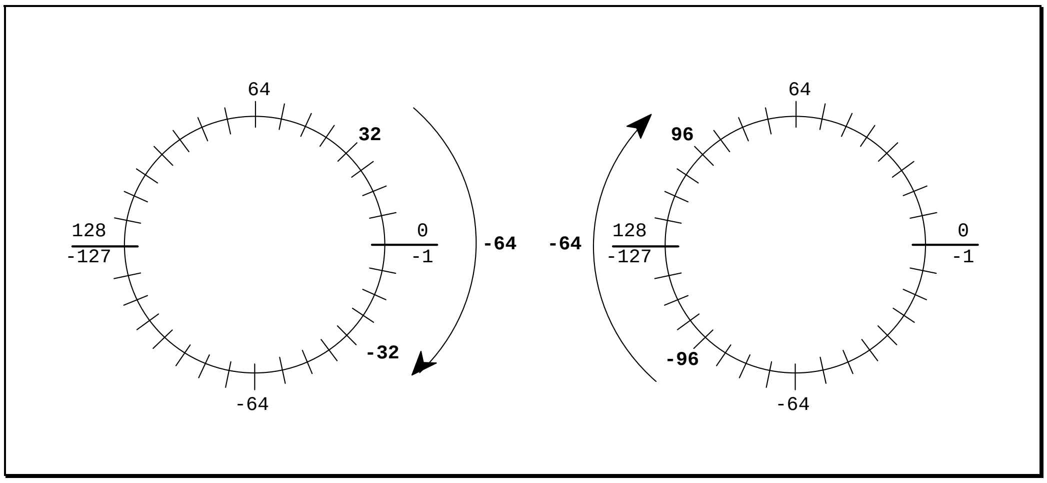


Figure 2.16 – 8-bit signed subtraction

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.

The overflow bit, V, is set after a signed add or subtract when the result is incorrect. (2.13)

Overflow can be detected by Boolean operations on the individual bits

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 *X*7-*X*0 and *M*7-*M*0 be the individual binary bits of the two 8-bit numbers which are to be added, and let *R*7-*R*0 be individual binary bits of the 8-bit sum. Then, the 4 condition code bits after an addition are shown in Table 2.38.

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	$X7\cdot M7\cdot\overline{R7}$ $+\overline{X7}\cdot\overline{M7}\cdot R7$	add two positives get a negative result; or add two negatives get a positive result
C	$X7\cdot M7$ $+M7\cdot\overline{R7}$ $+X7\cdot\overline{R7}$	add two numbers both above 127; or add one number above 127 and get a number below 128; or add one number above 127 and get a number below 128

Table 2.38 – 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 2.39.

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	$X7\cdot\overline{M7}\cdot\overline{R7}$ $+\overline{X7}\cdot M7\cdot R7$	a positive minus a negative and get a positive result; or a negative minus a positive and get a negative result
C	$\overline{X7}\cdot M7$ $+M7\cdot R7$ $+\overline{X7}\cdot R7$	a number below 128 minus a number above 127; or subtracted a number above 127 and get one above 127; or started with a number below 127 and get one above 127

Table 2.39 – Condition code bits after an 8-bit subtraction operation

Ignoring overflow (signed or unsigned) can result in significant errors. (2.14)

Computers have two sets of conditional branch instructions (if statements) which make program decisions based on either the `C` or `V` bit. (2.15)

An error will occur if you use unsigned conditional branch instructions (if statements) after operating on signed numbers, and vice-versa. (2.16)

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.

Promotion is used by compilers to avoid overflow and underflow problems

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, X_{16}, M_{16}, and R_{16} are 16-bit intermediate values, and R is an 8-bit unsigned output.

Promotion of unsigned numbers to avoid overflow and underflow

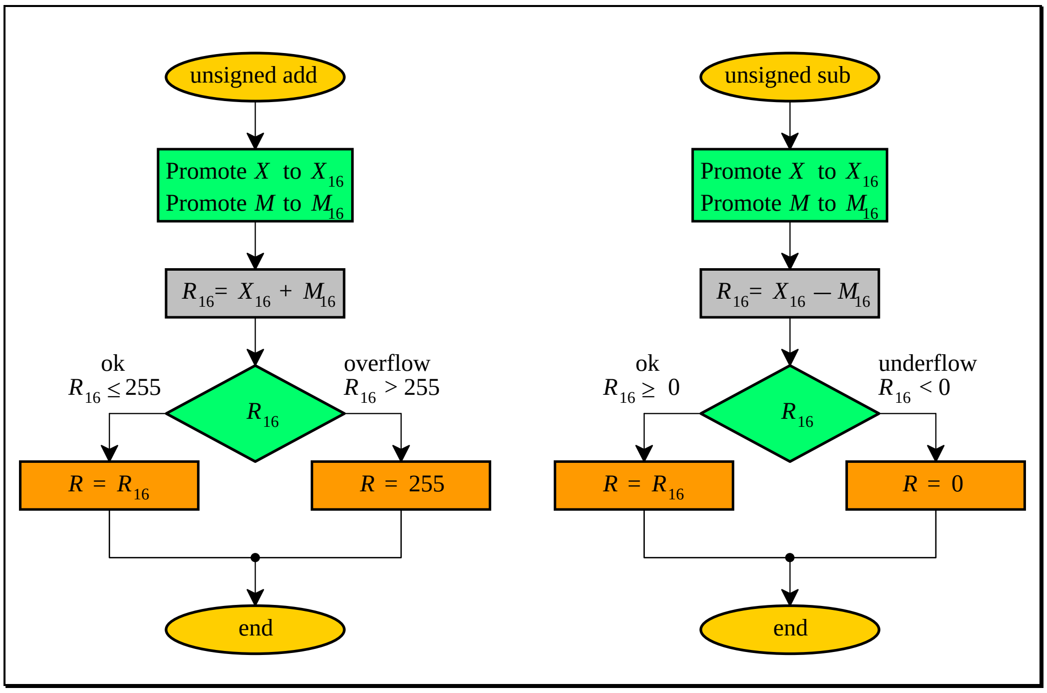


Figure 2.17 – Promotion can be used to avoid overflow and underflow

Signed numbers are promoted by extending the sign bit

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 $\underline{-24}$ -160	1010 0000 $\underline{-0100\ 0000}$ 0110 0000	1111 1111 1010 0000 $\underline{-0000\ 0000\ 0100\ 0000}$ 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, X_{16}, M_{16}, and R_{16} are 16-bit signed intermediate values, and R is an 8-bit signed output.

Promotion of signed numbers to avoid overflow and underflow

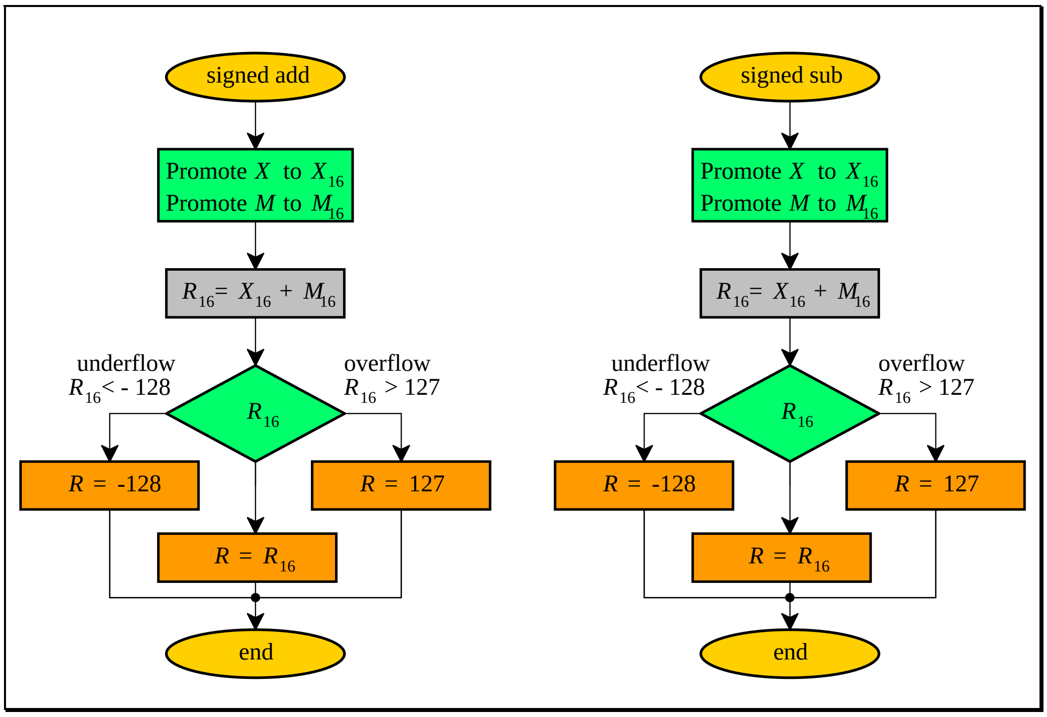


Figure 2.18 – Promotion can be used to avoid overflow and underflow

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.)

Ceiling and floor can be used to avoid overflow and underflow

Using ceiling and floor of unsigned numbers to avoid overflow and underflow

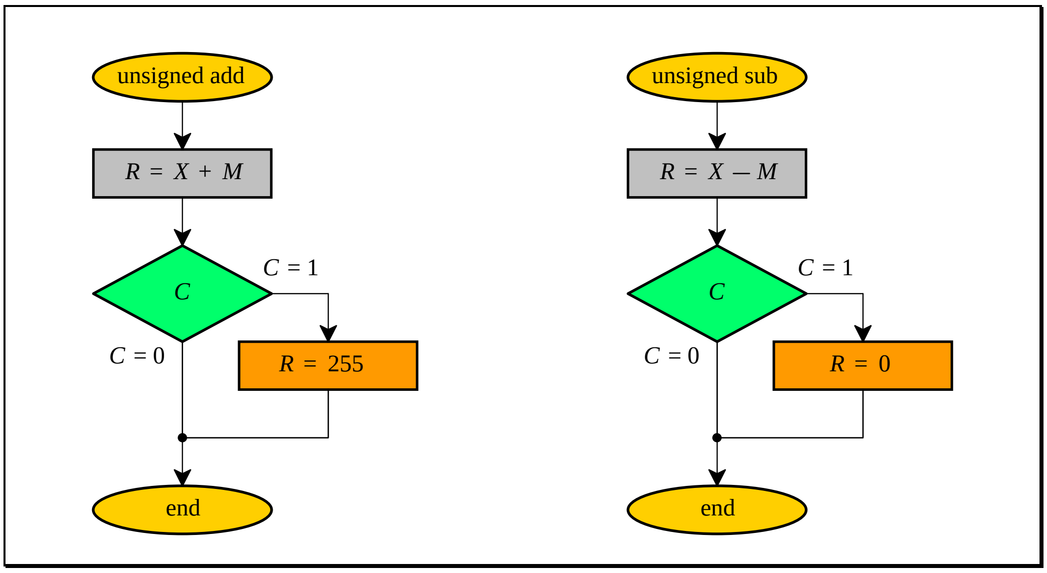


Figure 2.19 – In assembly language we can detect overflow and underflow

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.

Using ceiling and floor of signed numbers to avoid overflow and underflow

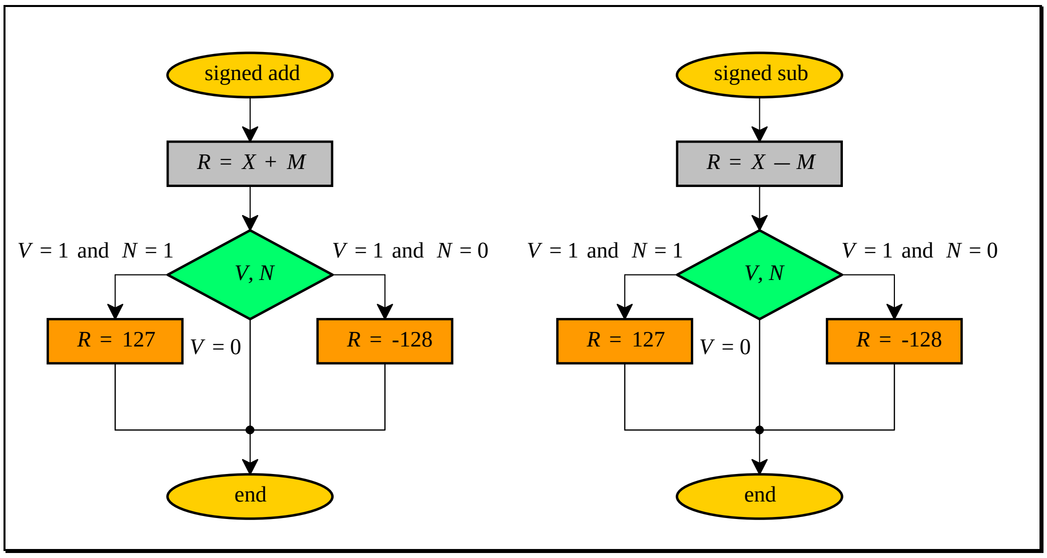


Figure 2.20 – In assembly language we can detect overflow and underflow

In summary, overflow and underflow occur *silently* during addition and subtraction of integer data types; there is no run-time checking provided by the microprocessor. It is entirely the programmer's responsibility to allocate a data type of the appropriate size for each variable and to avoid overflow.

Procedural Statements

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 section 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, 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 of 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.

Compound statements and decision statements are used to make algorithms of arbitrary complexity

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

Simple statements are terminated with a semicolon

```
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 the following example the local variable `temp` is only defined within the inner compound statement.

Compound statements are enclosed with braces { }

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

Listing 2.54 – Examples of 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. The **`goto`** and **`switch`** statements (below) could violate this rule.

The `if` Statement

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
```

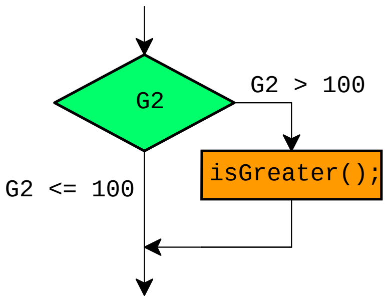
or

```
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 the following example, the function `isGreater()` is executed if `G2` is larger than `100`.

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



Listing 2.55 – Example `if` statement

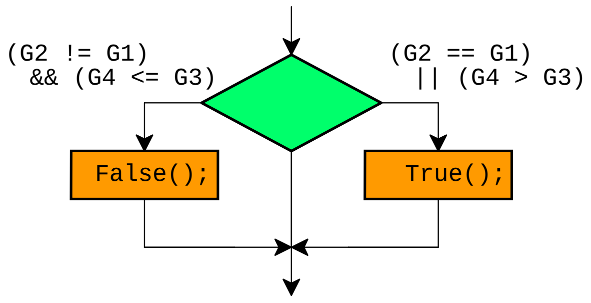
A 3-wide median filter can be designed using **`if-else`** conditional statements.

```
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 2.56 – A 3-wide median function

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

```
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

The **`switch`** statement

```
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.

The **`switch`** statement uses **`case`** and **`default`** prefixes

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

The **break** statement is used to exit a block

```
break;
```

Some examples may help clarify these ideas. Assume Port A is specified as an output, and bits 3, 2, 1, and 0 are connected to a stepper motor. The **switch** statement will first read Port A and AND the data with `0x0F` (`PORTA & 0x0F`). If the result is 5, then Port A 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 PORTA *(unsigned char volatile *)(0x0000)
void step(void)
{
  // turn stepper motor one step
  switch (PORTA & 0x0F)
  {
    case 0x05:
      PORTA = 0x06; // 6 follows 5
      break;
    case 0x06:
      PORTA = 0x0A; // 10 follows 6
      break;
    case 0x0A:
      PORTA = 0x09; // 9 follows 10
      break;
    case 0x09:
      PORTA = 0x05; // 5 follows 9
      break;
    default:
      PORTA = 0x05; // start at 5
   }
}
```

Listing 2.57 – Example of the switch statement

This next example shows that 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 2.58 – 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
```

The **`while`** statement

where `ExpressionList` is a list of one or more expressions and `Statement` is a 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 polling loops

```
#define RDRF 0x20 // Receive Data Register Full Bit
// Wait for new serial port input,
// return ASCII code for key typed
char SCI_InChar(void)
{
  while ((SCISR1 & RDRF) == 0);
  return (SCIDRL);
}

#define TDRE 0x80 // Transmit Data Register Empty Bit
// Wait for buffer to be empty, output ASCII to serial port
void SCI_OutChar(char data)
{
  while ((SCISR1 & TDRE) == 0);
  SCIDRL = data;
}
```

Listing 2.59 – 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

The **continue** statement causes control to jump to the top of the control loop

```
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

The **`for`** statement

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

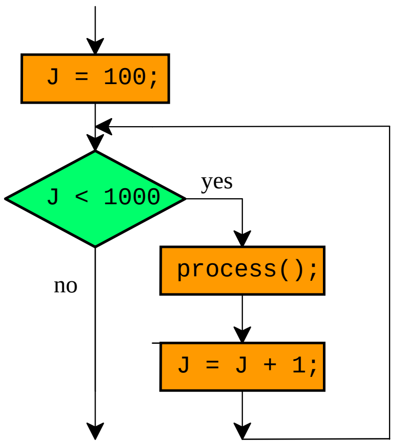
`for` statements are performed in the following steps:

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. For example,

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



A five-element array being 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. 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 ) ;
```

The **do** statement

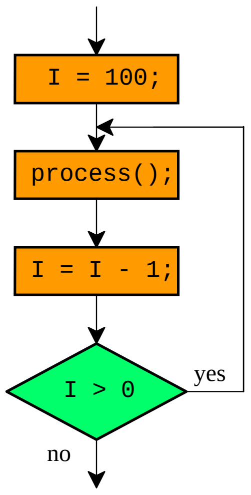
`Statement` is any simple or compound statement. The **`do`** statement executes in the following steps:

`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

The **`return`** statement

```
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. Statements that do nothing can serve a purpose. As we saw previously, 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.

Nulls statements (statements that do nothing) can serve a purpose in a program

Take, for example, the statement

```
while ((SCISR1 & TDRE) == 0); // Wait for TDRE to be set
```

in which the `((SCISR1 & TDRE) == 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

The **goto** statement…

```
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 2.60 – Examples of 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.

…should never be used

Because they violate the structured programming criteria, **goto** statements should not be used at all.

Missing Statements

The C language has no input/output, program control, or memory management statements. 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.

High-level functions like input / output are implemented in C libraries – this makes C very portable

Pointers

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

Pointers are variables that store addresses

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 all addresses are pointers

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 the next sections, an unsubscripted array name yields the address of the array, and 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. 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 except that pointers are distinguished by an asterisk that prefixes their names. Listing 2.61 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, using a small memory model with CodeWarrior creates pointers containing 16-bit unsigned absolute addresses. This means that the small memory model does not provide for direct support of the extended memory available on the MC9S12A512 microcontroller.

Pointers are declared by placing a * in front of the pointer name

```
// define pt1, declare as a pointer to a 16-bit integer
short *pt1;

// define pt2, declare as a pointer to an 8-bit character
char *pt2;

// define data and pt3, declare data as an unsigned 16-bit
// integer and declare pt3 as a pointer to a 16-bit unsigned
// integer
unsigned short data, *pt3;

// define pt4, declare as a pointer to a 32-bit integer
long *pt4;

// declare pt5 as a pointer to an integer
extern short *pt5;
```

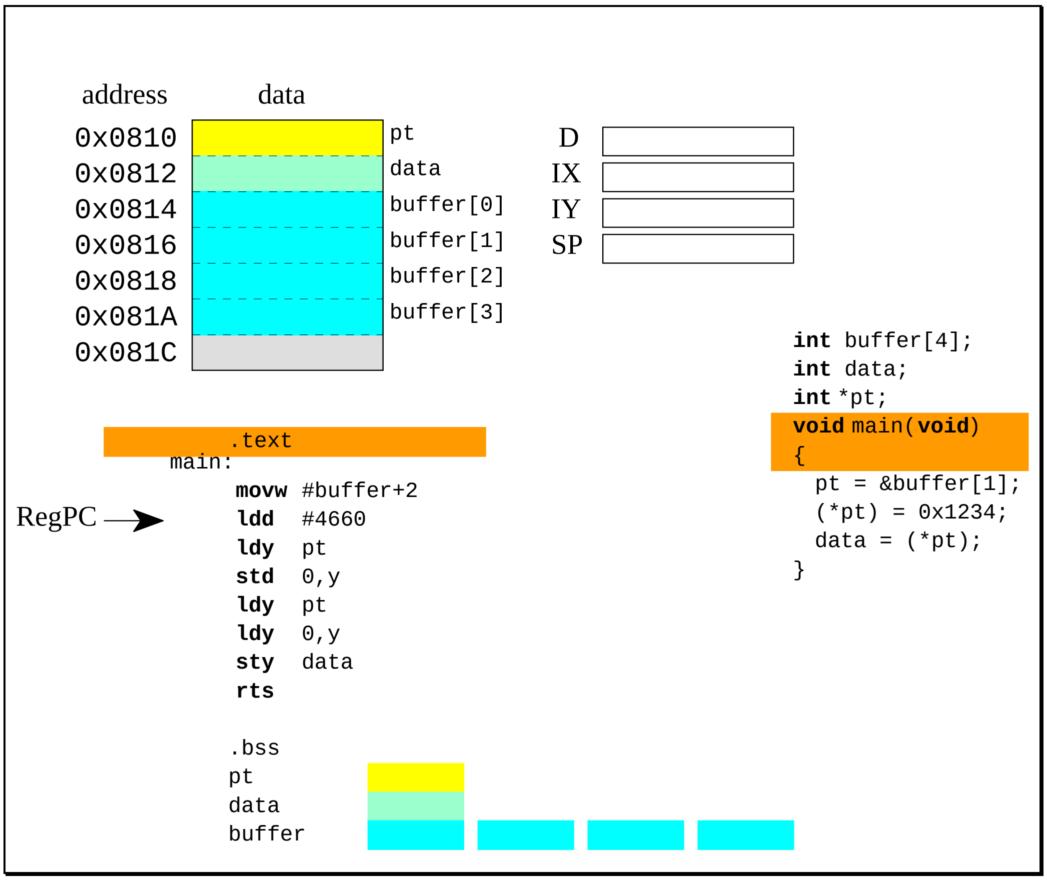
Listing 2.61 – Examples of pointer declarations

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 2.61 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. Figure 2.21 to Figure 2.28 illustrate several legitimate pointer references. In the first figure, the global variables contain unknown data (actually we know CodeWarrior will zero global variables). The arrow identifies the execution location. Assume addresses `0x0810` through `0x081A` exist in RAM.

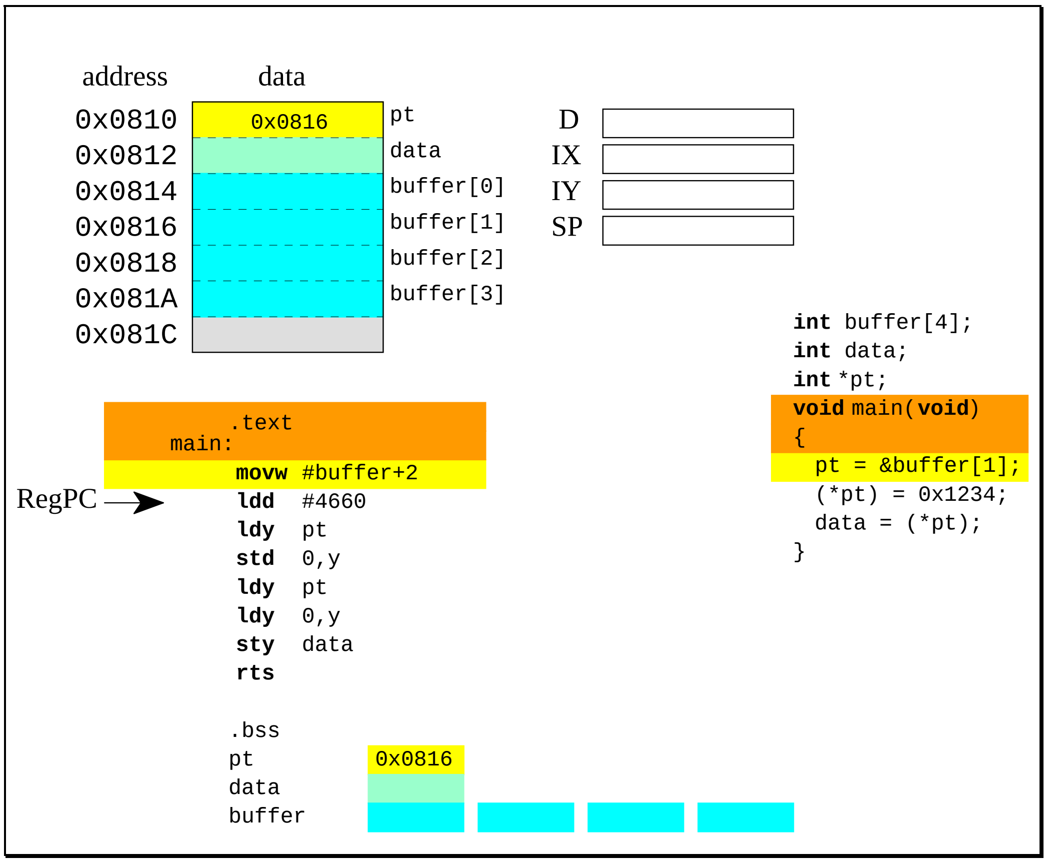
Example of legitimate pointer references in C



```
          .text
     main:
          movw  #buffer+2
          ldd   #4660
          ldy   pt
          std   0,y
          ldy   pt
          ldy   0,y
          sty   data
          rts

          .bss
          pt
          data
          buffer
```

```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

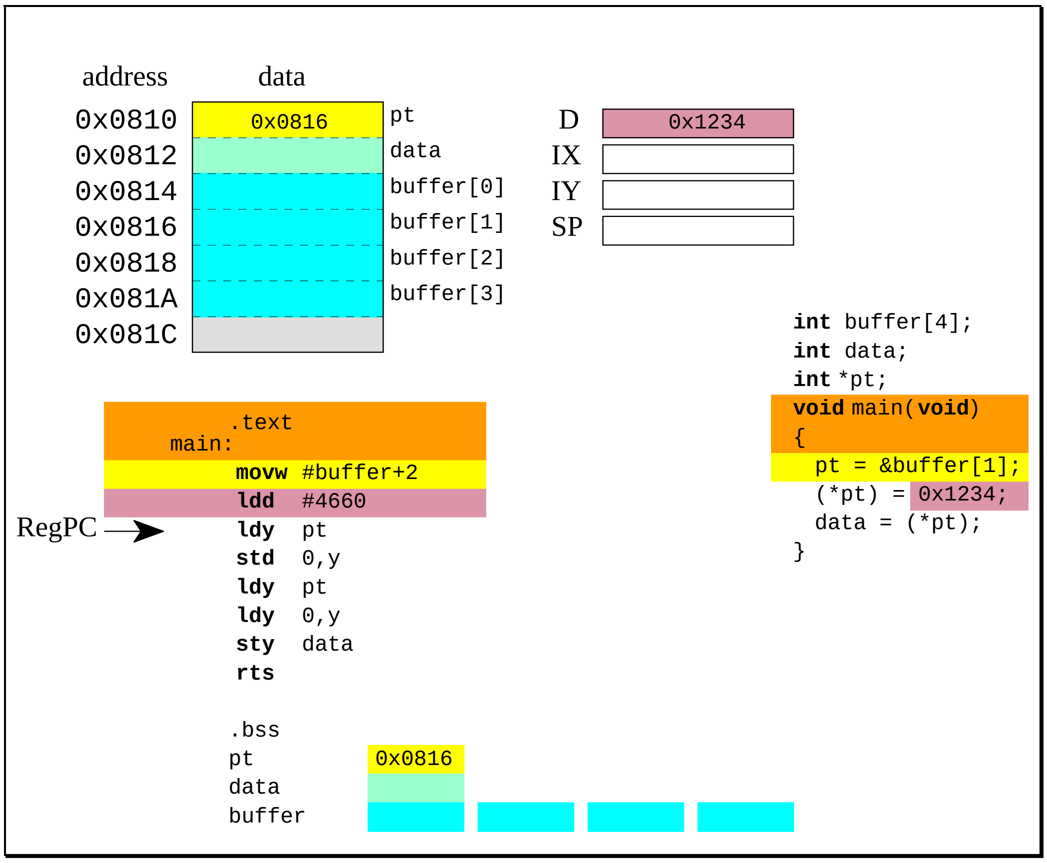
Figure 2.21 – MC9S12 example of pointer references – step 1



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
          .text
     main:
               movw  #buffer+2
               ldd   #4660
               ldy   pt
               std   0,y
               ldy   pt
               ldy   0,y
               sty   data
               rts

          .bss
          pt
          data
          buffer
```

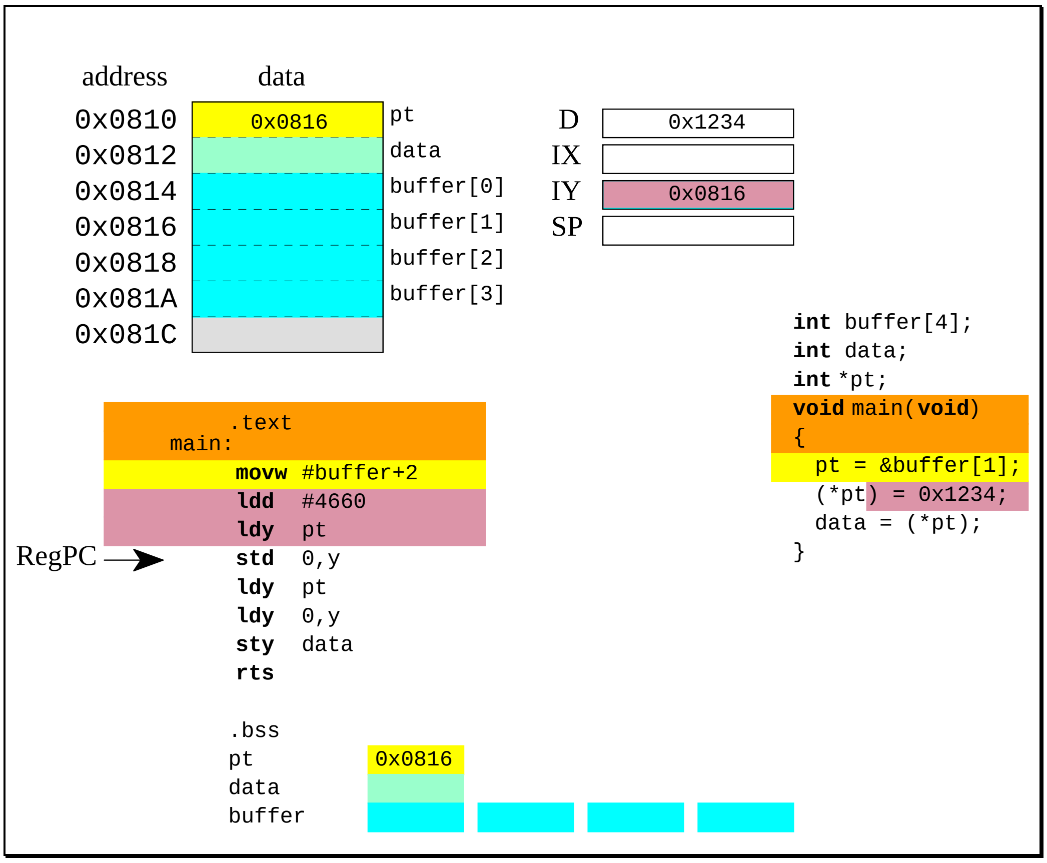
Figure 2.22 – MC9S12 example of pointer references – step 2



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
          .text
     main:
               movw  #buffer+2
               ldd   #4660
               ldy   pt
               std   0,y
               ldy   pt
               ldy   0,y
               sty   data
               rts

          .bss
          pt
          data
          buffer
```

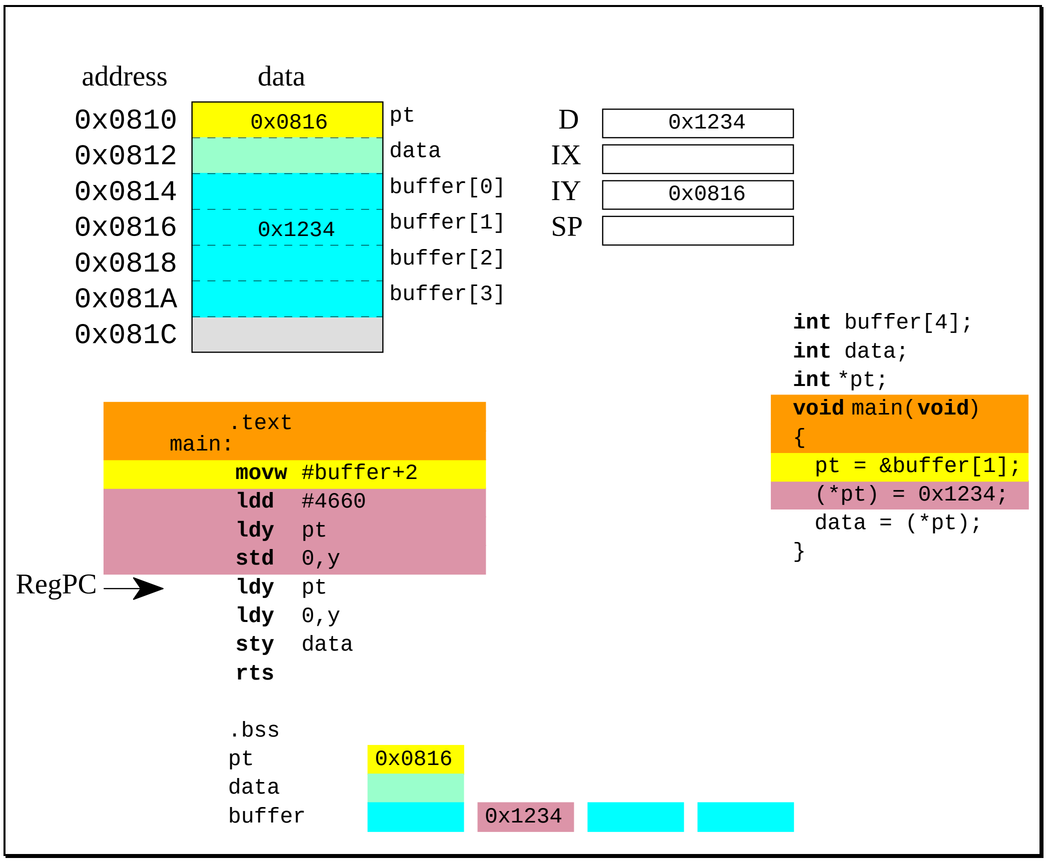
Figure 2.23 – MC9S12 example of pointer references – step 3



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
        .text
main:
        movw  #buffer+2
        ldd   #4660
        ldy   pt
        std   0,y
        ldy   pt
        ldy   0,y
        sty   data
        rts
```

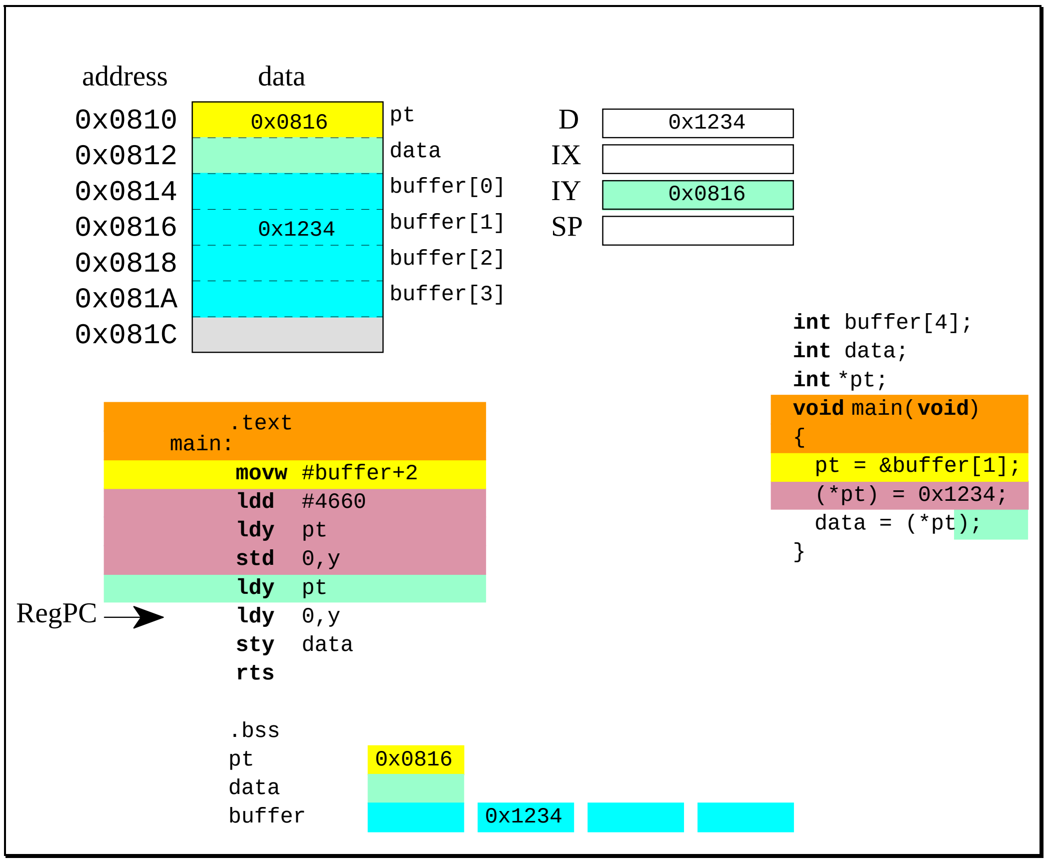
Figure 2.24 – MC9S12 example of pointer references – step 4



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
        .text
main:
        movw  #buffer+2
        ldd   #4660
        ldy   pt
        std   0,y
        ldy   pt
        ldy   0,y
        sty   data
        rts
```

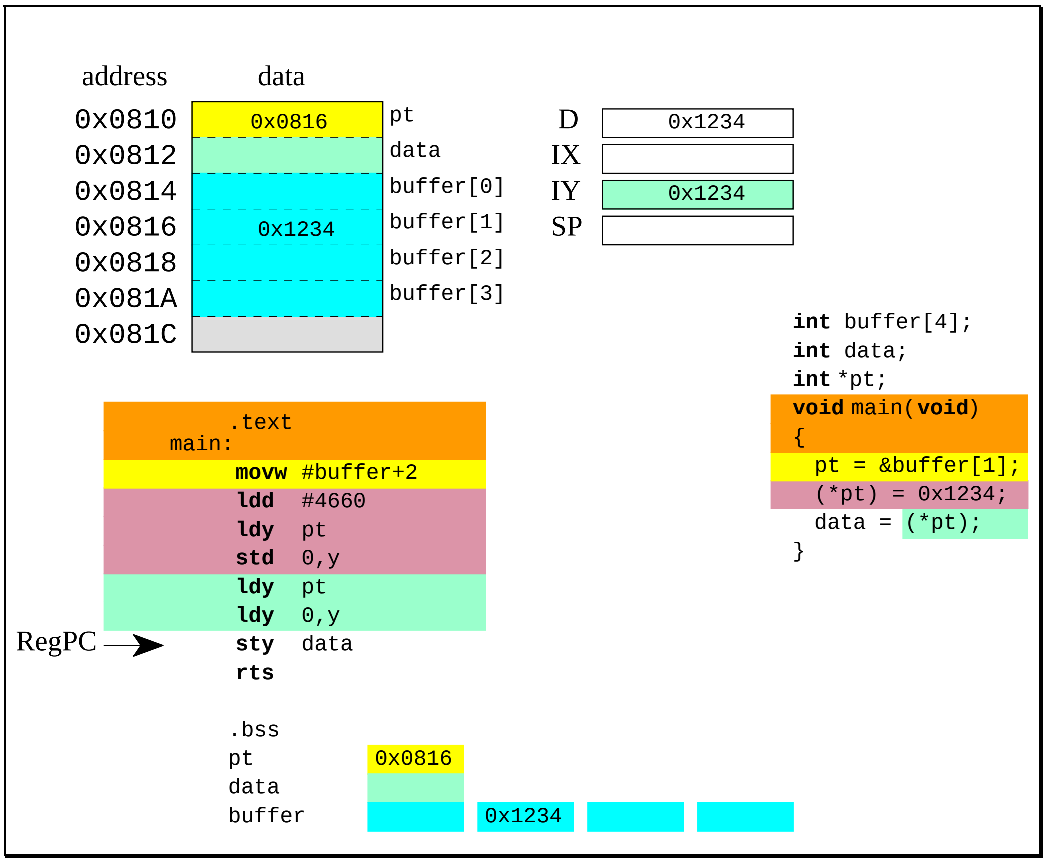
Figure 2.25 – MC9S12 example of pointer references – step 5



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
           .text
     main:
           movw #buffer+2
           ldd  #4660
           ldy  pt
           std  0,y
           ldy  pt
           ldy  0,y
           sty  data
           rts
```

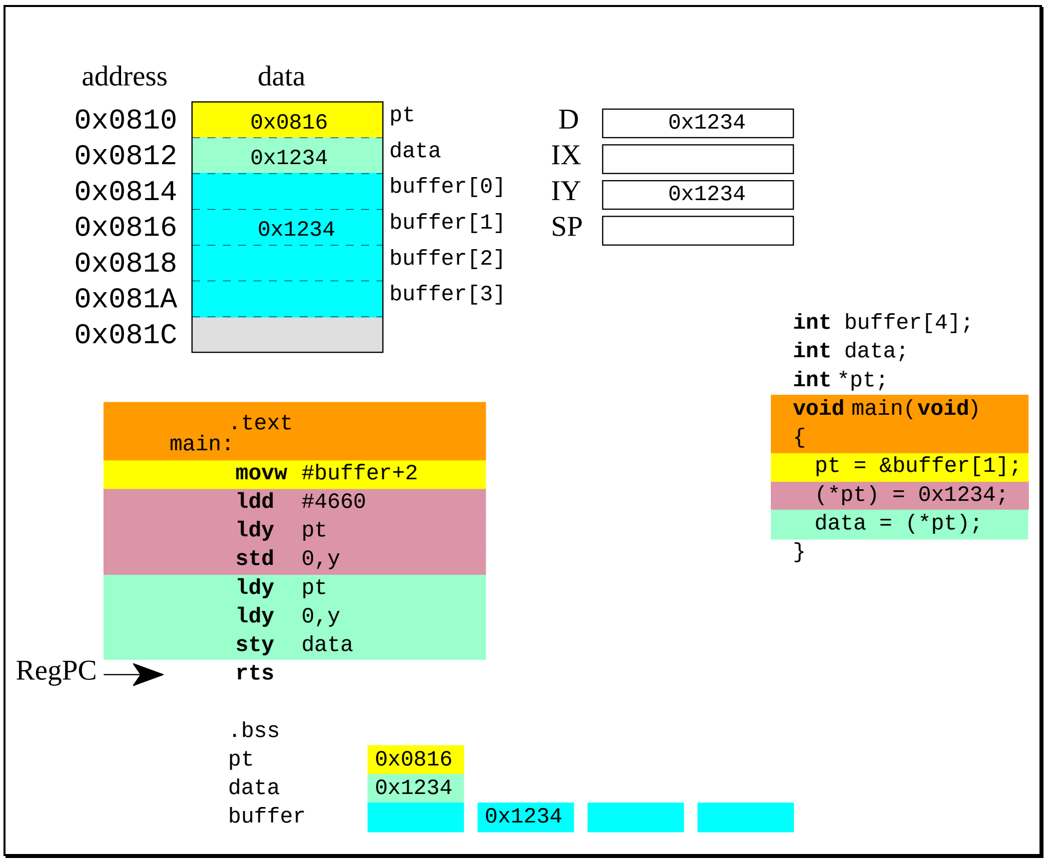
Figure 2.26 – MC9S12 example of pointer references – step 6



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
           .text
     main:
           movw #buffer+2
           ldd  #4660
           ldy  pt
           std  0,y
           ldy  pt
           ldy  0,y
           sty  data
           rts
```

Figure 2.27 – MC9S12 example of pointer references – step 7



```
int buffer[4];
int data;
int *pt;
void main(void)
{
  pt = &buffer[1];
  (*pt) = 0x1234;
  data = (*pt);
}
```

```
          .text
     main:
          movw #buffer+2
          ldd  #4660
          ldy  pt
          std  0,y
          ldy  pt
          ldy  0,y
          sty  data
          rts

          .bss
          pt
          data
          buffer
```

Figure 2.28 – MC9S12 example of pointer references – step 8

The expression `&buffer[1]` returns the address of the second 16-bit element of the buffer (`0x0816`). Therefore the line `pt=&buffer[1];` makes `pt` point to `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 16-bit signed integer at `0x0816`". You can optionally add the parentheses `()` to clarify that `*` and `pt` are one object. In this case the parentheses are not needed. Later when we perform address arithmetic, the parentheses will be important. Therefore the line `(*pt)=0x1234;` sets `buffer[1]` to `0x1234`.

When the `*pt` occurs on the right-hand-side of an assignment statement, data is retrieved from memory at the address. Again, you can optionally 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 16-bit information from `buffer[1]` into `data`).

We can get a better understanding of pointers by observing the assembly generated by our compiler. The following MC9S12 assembly was generated by CodeWarrior when the above pointer example (Figure 2.28) was compiled. Notice that the CodeWarrior compiler is highly optimized.

```
main:
    movw  #buffer:2,pt ;pt = &buffer[1];
    ldd   #4660        ;(*pt) = 0x1234;
    std   [pt,PCR]
    std   data         ;data = (*pt);
    rts
```

Listing 2.62 – Examples of pointer references optimised by CodeWarrior

Memory Addressing

The size of a pointer depends on the architecture of the CPU and the implementation of the C compiler. The MC9S12 employs an absolute memory addressing scheme in which an effective address is composed simply of a single 16-bit unsigned value. In particular the MC9S12 registers are shown in Figure 2.29. The MC9S12512 does provide for extended addressing. For more information on this feature, you need to refer to the Freescale documentation.

With a small memory model, addressing is "flat" and occurs with 16-bit addresses

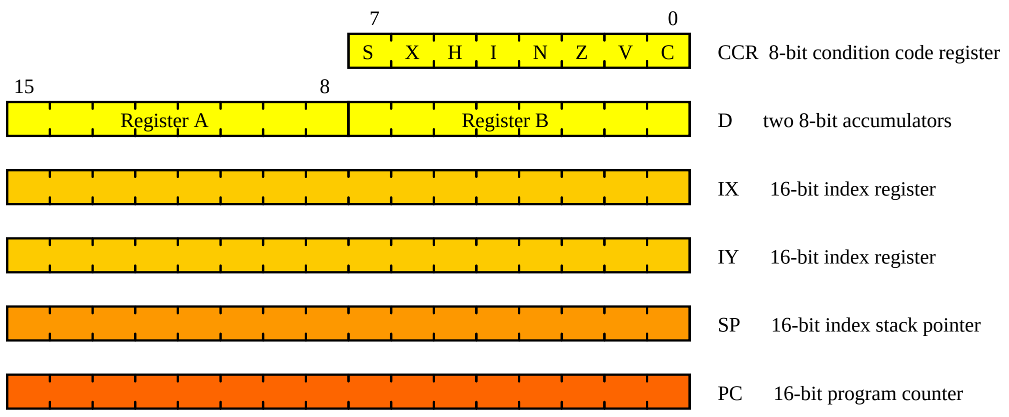


Figure 2.29 – The MC9S12 has 16-bit address registers, `IX`, `IY`, `SP` and `PC`

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. The other types are usually loaded with object code from our S19 file and our program is allowed only to read the data. Table 2.40 shows the various types of memory available in the MC9S12 microcontroller. 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. EEPROM is a technology that allows individual small sectors (typically 4 bytes) to be erased and bytes individually written. Most microcontrollers now have non-volatile Flash ROM as the main program memory, which has bulk erasure (typically 16 kB) and individual write capability at the byte level. The one-time-programmable (OTP) ROM is a simple non-volatile storage technology used in large volume products that can be programmed only once by the semiconductor manufacturer.

Types of memory in the MC9S12

Memory	**When power is removed**	**Ability to Read/Write**	**Program cycles**
RAM	volatile	Random and fast access	infinite
Battery-backed RAM	non-volatile	Random and fast access	infinite
EEPROM	non-volatile	Easily reprogrammed	300 000 times
Flash	non-volatile	Easily reprogrammed	100 000 times
OTPROM	non-volatile	Can be programmed once at the factory	N/A

Table 2.40 – Various types of memory available for the MC9S12

From a logical standpoint we implement segmentation when we group together in memory information that has similar properties or usage. Typical software segments include global variables (`.data` section), the heap, local variables, fixed constants (`.rodata` section), and machine instructions (`.text` section). Global variables are permanently allocated and usually accessible by more than one program. We must use global variables for information that must be permanently available, or for information that is to be shared by more than one module. We will see the first-in-first-out (FIFO) queue is a global data structure that is shared by more than one module. CodeWarrior allows the use of a heap to dynamically allocate and release memory. This information can be shared or not shared depending on which modules have pointers to the data. The heap is efficient in situations where storage is needed for only a limited amount of time. Local variables are usually allocated on the stack at the beginning of the function, used within the function, and deallocated at the end of the function. Local variables are not shared with other modules. Fixed constants do not change and include information such as numbers, strings, sounds and pictures. Just like the heap, the fixed constants can be shared or not shared depending on which modules have pointers to the data.

The type of memory dictates its usage

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 non-volatile memory. If there is both EEPROM and Flash on our microcontroller, we put some fixed constants in EEPROM and some in Flash. 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. For a project with a large volume it will be cost effective to place the machine instructions in OTPROM.

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 is done automatically by the compiler, and 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 we define structures of other sizes, the appropriate factors would have to be used.

Pointer arithmetic takes into account the size of the data being pointed to

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 the next section for more information 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` also points to a signed 16-bit integer.

Type is preserved in pointer arithmetic

Precedence determines the order of evaluation. One of the most common mistakes results when the programmer neglects 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)`. Remember (2.2): "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

Pointers are always unsigned

```
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 2.63 – Example showing a pointer comparison

which performs an unsigned comparison since `pt1` and `pt2` are pointers. Thus, if `pt2[5]` contains `0xF000` and `pt1` contains `0x1000`, the expression will yield false, since `0xF000` is a higher unsigned value than `0x1000`.

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.

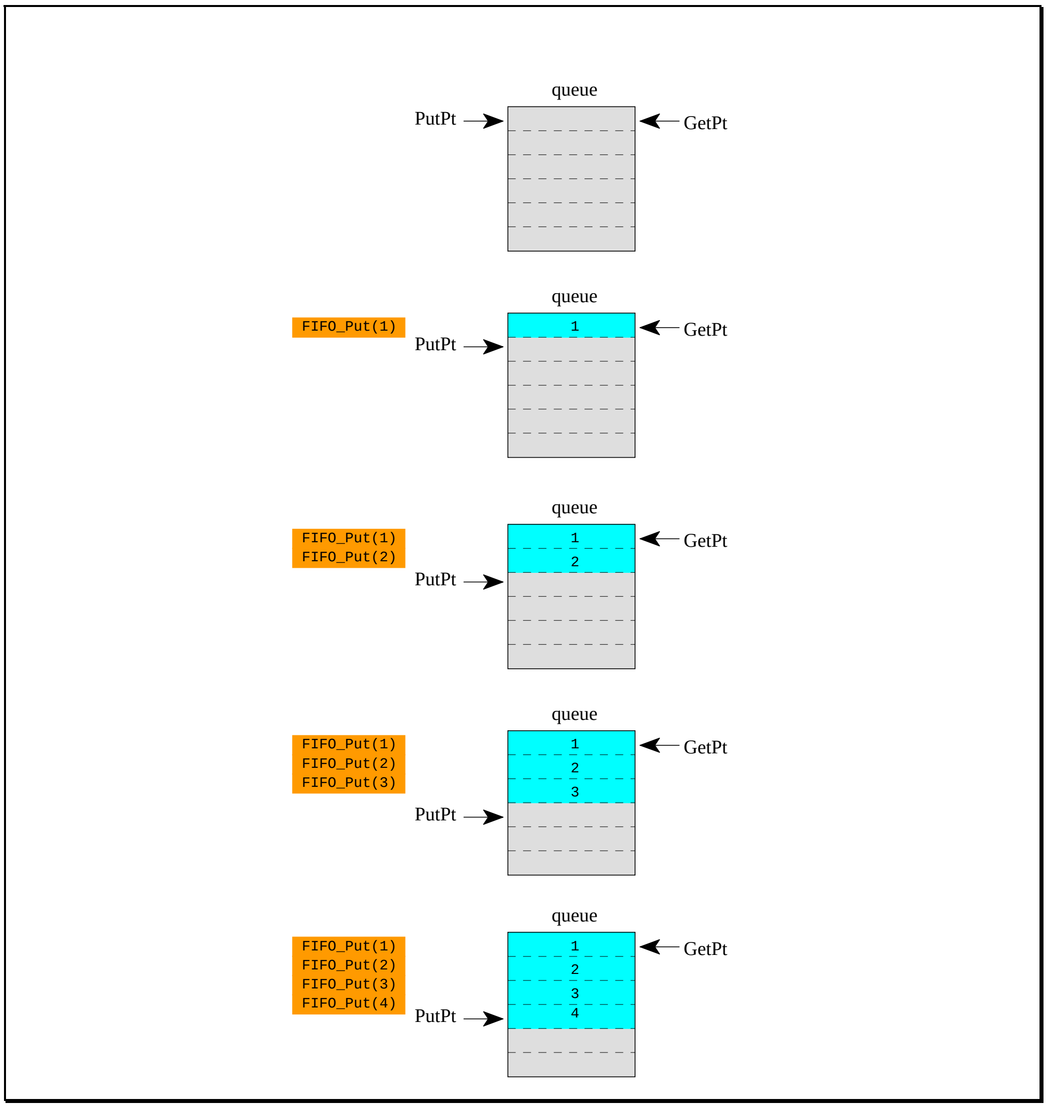
The address of zero is reserved for **`NULL`** – a pointer that doesn't yet point to anything

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

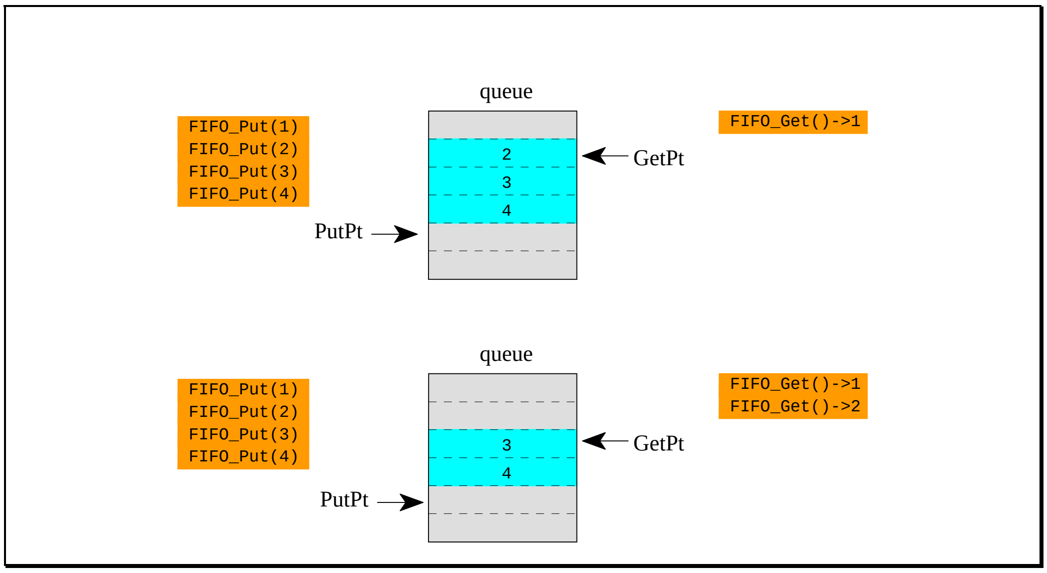
To illustrate the use of pointers we will design a two-pointer FIFO. The first-in first-out circular queue (FIFO) is also useful for data flow problems. It is a very common data structure used for I/O interfacing. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs 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 `FIFO_Get()`, and `PutPt` points to the empty space where the data will stored by the next call to `FIFO_Put()`. If the FIFO is full when `FIFO_Put()` is called then the subroutine should return a full error (e.g., `V=1`). Similarly, if the FIFO is empty when `FIFO_Get()` is called, then the subroutine should return an empty error (e.g., `V=1`). The `PutPt` and `GetPt` pointers must be wrapped back up to the top when they reach the bottom.



Four FIFO_Put() operations…

Figure 2.30 – FIFO example showing the wrapping of pointers – step 1



Two FIFO_Get() operations…

Figure 2.31 – FIFO example showing the wrapping of pointers – step 2

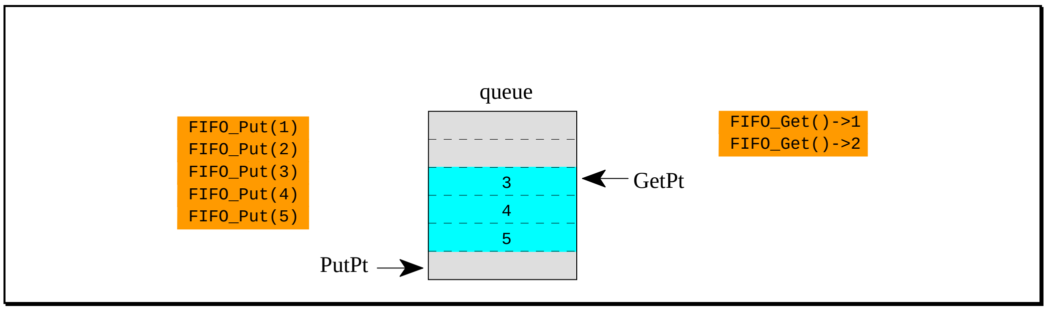


Figure 2.32 – FIFO example showing the wrapping of pointers – step 3

A FIFO_Put() followed by a FIFO_Get()

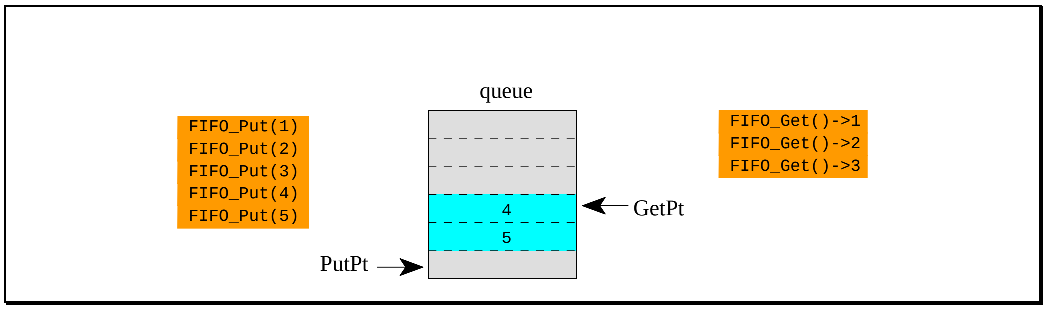


Figure 2.33 – FIFO example showing the wrapping of pointers – step 4

Two FIFO_Put() operations…

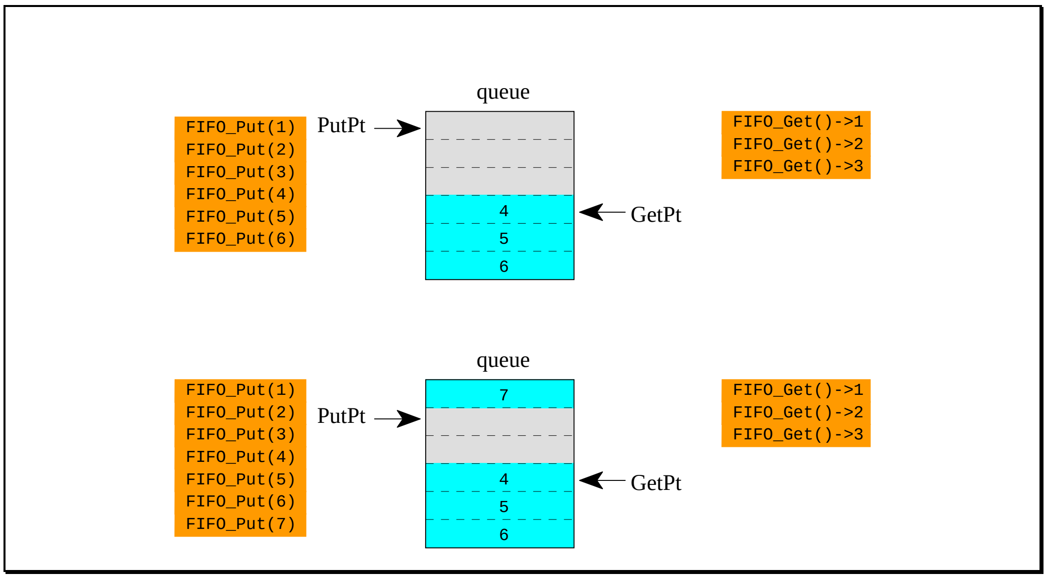
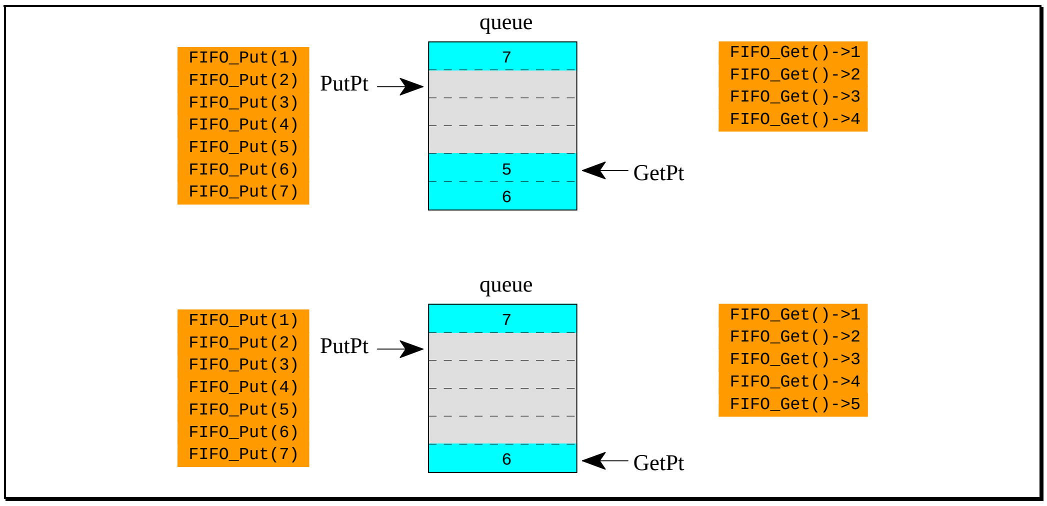
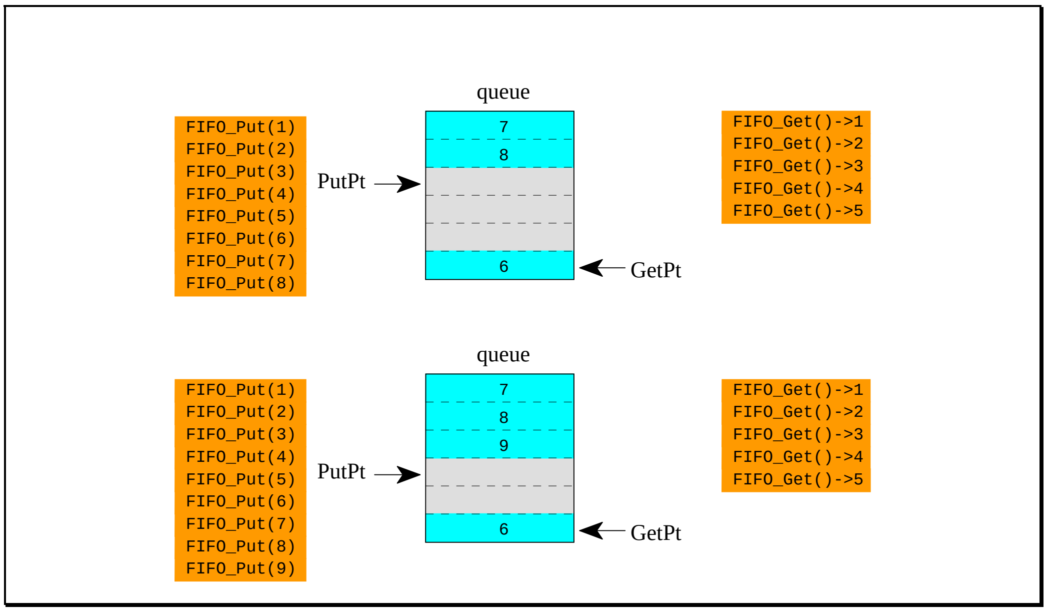


Figure 2.34 – FIFO example showing the wrapping of pointers – step 5



Two FIFO_Get() operations…

Figure 2.35 – FIFO example showing the wrapping of pointers – step 6



Two FIFO_Put() operations…

Figure 2.36 – FIFO example showing the wrapping of pointers – step 7

Finally, four FIFO_Get() operations that empty the queue

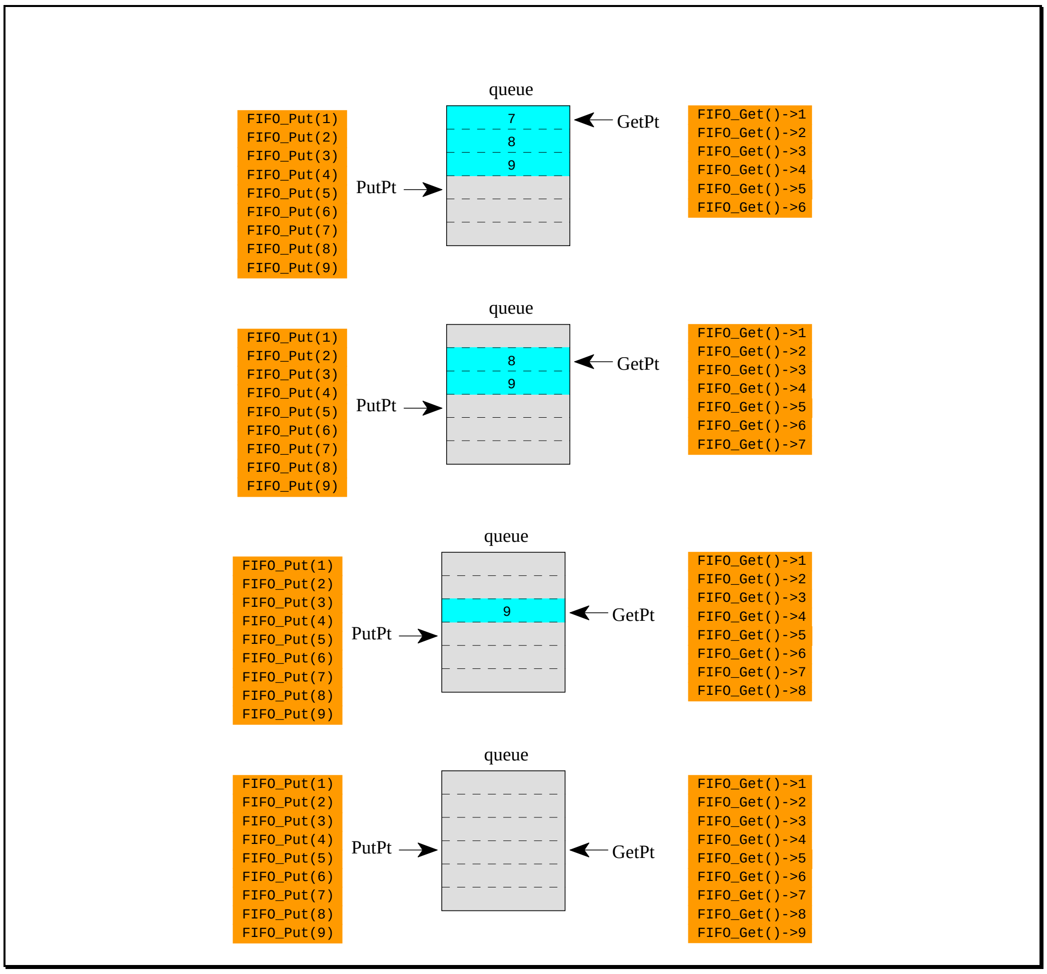


Figure 2.37 – FIFO example showing the wrapping of pointers – step 8

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. `FIFO_Get()` would decrement the counter and `FIFO_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 `FIFO_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 `FIFO_Get()`, then the FIFO is empty. Similarly, if `PutPt + 1` equals `GetPt` at the beginning of `FIFO_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 FIFO_SIZE 10 // Max number of 8-bit data in the FIFO
#define START_CRITICAL() { asm tfr ccr,a; asm staa savedCCR; asm sei; }
#define END_CRITICAL()   { asm ldaa savedCCR; asm tfr a,ccr; }

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[FIFO_SIZE]; // The statically allocated FIFO data

void FIFO_Init(void)
{
  unsigned char savedCCR;
  // make atomic, entering critical section
  START_CRITICAL();
  PutPt = GetPt = &Fifo[0]; // Empty when PutPt == GetPt
  END_CRITICAL();  // end critical section
}

int FIFO_Put(char data)
{
  char *pt; // Temporary put pointer
  unsigned char savedCCR;
  // make atomic, entering critical section
  START_CRITICAL();
  pt = PutPt;     // Copy of put pointer
  *(pt++) = data; // Try to put data into FIFO
  if (pt == &Fifo[FifoSize])
    pt = &Fifo[0]; // Wrap
  if (pt == GetPt)
  {
    END_CRITICAL();  // end critical section
    return (0);
  }   // Failed, FIFO was full
  else
  {
    PutPt = pt;
    END_CRITICAL();  // end critical section
    return (-1);     // Successful
  }
}

int FIFO_Get(char *dataPt)
{
  unsigned char savedCCR;
  if (PutPt == GetPt)
  {
    return (0);
  }   // Empty if PutPt == GetPt
  else
  {
    // make atomic, entering critical section
    START_CRITICAL();
    *dataPt = *(GetPt++);
    if (GetPt == &Fifo[FifoSize])
      GetPt = &Fifo[0];
    END_CRITICAL();  // end critical section
    return (-1);
  }
}
```

Listing 2.64 – FIFO queue implemented with pointers

The `START_CRITICAL()` macro is defined to save the state of the global interrupt enable bit and disable interrupts. This prevents another thread from interfering with the FIFO operation. The `END_CRITICAL()` macro restores the state of the global interrupt enable bit.

You have to be careful when multiple threads are using the same resource

Since these routines have read / modify / write accesses to global variables the three functions (`FIFO_Init()`, `FIFO_Put()`, `FIFO_Get()`) are themselves not re-entrant. Consequently interrupts are temporarily disabled, to prevent one thread from re-entering these FIFO functions. One advantage of this pointer implementation is that if you have a single thread that calls `FIFO_Get()` (e.g., the main program) and a single thread that calls `FIFO_Put()` (e.g., the serial port receive interrupt handler), then this `FIFO_Put()` function can interrupt this `FIFO_Get()` 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 `FIFO_Get()` (e.g., the serial port transmit interrupt handler) and a single thread calling `FIFO_Put()` (e.g., the main program.) On the other hand, if the situation is more general, and multiple threads could call `FIFO_Put()` or multiple threads could call `FIFO_Get()`, then the interrupts would have to be temporarily disabled as shown.

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 section because it involves addresses. The format used by the CodeWarrior compiler fits the following model. The following listing shows one 8-bit and two 16-bit MC9S12 I/O ports. The line `TFLG1 = 0x20;` generates an 8-bit I/O write operation to the port at address `0x004E`. The `TCNT` on the right hand side of the assignment statement generates a 16-bit I/O read operation from the port at address `0x0044`. The `TC5` on the left hand side of the assignment statement generates a 16-bit I/O write operation from the port at address `0x005A`. The `TFLG1` inside the **`while`** loop generates repeated 8-bit I/O read operations until bit 5 is set.

```
#define TFLG1 *(unsigned char volatile *)(0x004E)
#define TCNT *(unsigned short volatile *)(0x0044)
#define TC5 *(unsigned short volatile *)(0x005A)

void wait(unsigned short delay)
{
  TFLG1 = 0x20; // clear C5F
  TC5 = TCNT + delay; // TCNT at end of wait
  while ((TFLG1 & 0x20) == 0); // wait for C5F
}
```

Listing 2.65 – Sample CodeWarrior program that accesses I/O ports

It was mentioned earlier that the **`volatile`** modifier will prevent the compiler from optimizing I/O statements, i.e., these examples would not work if the compiler read `TFLG1` once, then used the same data over and over inside the **`while`** loop.

To understand this syntax we break it into parts. Starting on the right is the absolute address of the I/O port. For example the MC9S12 `TFLG1` register is at location `0x004E`. The parentheses are necessary because the definition might be used in an arithmetic calculation. For example the following two lines are quite different:

```
TheTime = *(unsigned char volatile *)(0x1023) + 100;
TheTime = *(unsigned char volatile *)0x1023 + 100;
```

In the second (incorrect) case the addition `0x01023 + 100` is performed on the address, not the data. The next part of the definition is a type casting. C allows you to change the type of an expression. For example (**`unsigned char volatile *`**) specifies that `0x1023` is an address that points at an 8-bit **`unsigned char`**. The `*` at the beginning of the definition causes the data to be fetched from the I/O port if the expression exists on the right-hand side of an assignment statement. The `*` also causes the data to be stored at the I/O port if the expression is on the left-hand side of the assignment statement. In this last way, I/O port accesses are indeed similar to pointers.

For example the previous example could have been implemented as:

```
unsigned char volatile *pTFLG1;
unsigned short volatile *pTCNT;
unsigned short volatile *pTC5;

void wait(unsigned short delay)
{
  pTFLG1 = (unsigned char volatile *)(0x004E);
  pTCNT = (unsigned short volatile *)(0x0044);
  pTC5 = (unsigned short volatile *)(0x005A);
  (*pTFLG1) = 0x20;
  (*pTC5) = (*pTCNT) + delay;
  while (((*pTFLG1) & 0x20) == 0);
}
```

Listing 2.66 – C program that accesses I/O ports using pointers

This function first sets the three I/O pointers then accesses the I/O ports indirectly through the pointers.

There is a problem when using pointer variables to I/O ports on the MC9S12. The `NULL` pointer typically is defined as address `0`, and `PORTA` also has address `0`.

Arrays and Strings

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 section apply equally to data arrays and character strings. String literals were discussed earlier; in this section 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 assign any 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 and 6th column into the variable `ThePosition`. If the array has three dimensions, then three subscripts are specified when referencing. Again we may assign any 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 16-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 exists 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 2.67 are valid declarations:

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

Listing 2.67 – Example showing 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 previous sections, 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 or 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. In this case, 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. Recall that the address operator & 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 2.68 – Example showing array references

Pointers and Array Names

The previous examples 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 2.69 – Example showing pointers to access array elements

It is important to realize that although C accepts unsubscripted array names as addresses, they are not the same as pointers. In the following example, we cannot 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 2.70 – 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 an ***lvalue***; i.e. 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 CalcSecondDerivative(void)
{
  short d2Vdt2;

  for (pt = data + 1; pt < data + 99; pt++)
     d2Vdt2 = (pt[-1] – 2 * pt[0] + pt[1]);
}
```

Listing 2.71 – 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, 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 the previous section on pointers, 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`

CodeWarrior implements 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.

```
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 2.72 – 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 n);
```

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 at 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);
int strcoll(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`. In general C allows the comparison rule used in `strcoll` to depend on the current locale, but in CodeWarrior `strcoll` is the same as `strcmp`.

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

We assume `src` 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 7 characters are the same:

```
n = strncmp("MC9S12A256", " MC9S12A512", 7);
```

The following function is similar to `strcpy`.

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

We assume `src` 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: ldaa 10,x ;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`. 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 `pt1` to point to the '`a`' in `label` and `pt2` to point to the second '`a`' in `ldaa`:

```
pt1 = strchr("label: ldaa 10,x ;comment", 'a');
pt2 = strrchr("label: ldaa 10,x ;comment", 'a');
```

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.

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

The function call will return 6 because there is a valid 6-digit hexadecimal string at the start of the line.

```
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 call sets `pt` to point to the '`l`' in `ldaa`:

```
pt = strstr("label: ldaa 10,x ;comment", "ldaa");
```

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 `FIFO_SIZE`), 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 returned 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.

```
// Index, counter implementation of the FIFO
#define FIFO_SIZE 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 == FIFO_SIZE
char FIFO[FIFO_SIZE];   // The statically allocated data

void FIFO_Init(void)
{
  PutI = GetI = Size = 0;    // Empty when Size==0
}

int FIFO_Put (char data)
{
  if (Size == FIFO_SIZE)
    return(0);              // Failed, FIFO was full

  Size++;
  FIFO[PutI++] = data;      // Put data into FIFO
  if (PutI == FIFO_SIZE)
    PutI = 0;               // Wrap
  return(-1);               // Successful
}

int FIFO_Get(char *datapt)
{
  if (Size == 0 )
    return(0);              // Empty if Size == 0

  *datapt = FIFO[GetI++];   // Get data out of FIFO
  Size--;
  if (GetI == FIFO_SIZE)
    GetI = 0;               // Wrap
  return(-1);               // Successful
}
```

Listing 2.73 – FIFO implemented with two indices and a counter

Structures

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 16-bit integer and two character pointers:

```
struct theport
{
  // 0 for I/O, 1 for in only, -1 for out only
  int mode;
  // pointer to its address
  unsigned char volatile *addr;
  // pointer to its data direction register
  unsigned char volatile *ddr;
};
```

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, PortC;
```

The previous line defines the three variables and allocates 6 bytes for each variable. If you knew you needed just three copies of structures of this type, you could have defined them as:

```
struct theport
{
  int mode;
  unsigned char volatile *addr;
  unsigned char volatile *ddr;
} PortA, PortB, PortC;
```

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
{
  int mode;      // 0 for I/O, 1 for in only -1 for out only
  unsigned char volatile *addr;  // address
  unsigned char volatile *ddr;   // data direction register
};

typedef struct theport port;

port PortA, PortB, PortC;
```

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

```
typedef struct
{
  int mode;      // 0 for I/O, 1 for in only -1 for out only
  unsigned char volatile *addr;  // address
  unsigned char volatile *ddr;   // data direction register
} port;

port PortA, PortB, PortC;
```

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:

```
PortA.mode = -1;      // Specify Port A as output
PortA.addr = (unsigned char volatile *)(0x0000);
PortA.ddr = (unsigned char volatile *)(0x0002);
(*PortA.ddr) = 0;

PortB.mode = 0;       // Port B is input and output
PortB.addr = (unsigned char volatile *)(0x0001);
PortB.ddr = (unsigned char volatile *)(0x0003);

(*PortA.addr) = (*PortB.addr);  // Copy from PortB to PortA
```

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

```
typedef struct
{
  int x1, y1;     // starting point
  int x2, y2;     // starting point
  char color;     // color
} line;

typedef struct
{
  line L1, L2;    // two lines
  char direction;
} path;

path p;           // global

void Setup(void)
{
  line myLine;
  path 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={5, 6, 10, 12, 255};  // black line from 5,6 to 10,12
  p.direction = -1;
  myLine = p.L1;
  q = {{0, 0, 5, 6, 128}, {5, 6, -10, 6, 128}, 1};
  q = p;
}
```

Listing 2.74 – 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 even numbers will actually allocate 10 and 20 bytes respectively. In particular, the MC9S12 executes faster out of external memory if 16-bit accesses occur on even addresses. For example, a 16-bit data access to an external odd address requires two bus cycles, while a 16-bit data access to an external even address requires only one bus cycle. There is no particular odd-address speed penalty for MC9S12 internal addresses (internal RAM or EEPROM). 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 19 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 thePath = {{0, 0, 5, 6, 128}, {5, 6, -10, 6, 128}, 1};
line theLine = {0, 0, 5, 6, 128};
port PortE =
{ 1,
  (unsigned char volatile *)(0x100A),
  (unsigned char volatile *)(0)
};
```

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

```
path thePath = {{0, 0, 5, 6, 128}, };
line theLine = {5, 6, 10, 12, };
port PortE = {1, (unsigned char volatile *)(0x100A), };
```

To place a structure in Flash memory, we define it as a global constant. In the following example the structure `fsm[3]` will be allocated and initialized in Flash memory. The linked structure of a finite state machine is a good example of a Flash-based structure.

```
typedef const struct State
{
  unsigned char out;            // Output to Port H
  unsigned short wait;          // Time (bus cycles) to wait
  unsigned char andMask[4];
  unsigned char equMask[4];
  const struct State *next[4]; // Next states
} TState;

typedef TState *PState;
```

```
#define Stop &FSM[0]
#define Turn &FSM[1]
#define Bend &FSM[2]

TState FSM[3] =
{
  {
     0x34,    2000,    // stop 1 ms
    {0xFF,    0xF0,    0x27,    0x00},
    {0x51,    0xA0,    0x07,    0x00},
    {Turn,    Stop,    Turn,    Bend}
  },
  {
     0xB3,    5000,    // turn 2.5 ms
    {0x80,    0xF0,    0x00,    0x00},
    {0x00,    0x90,    0x00,    0x00},
    {Bend,    Stop,    Turn,    Turn}
  },
  {
     0x75,    4000,    // bend 2 ms
    {0xFF,    0x0F,    0x01,    0x00},
    {0x12,    0x05,    0x00,    0x00},
    {Stop,    Stop,    Turn,    Stop}
  }
};
```

Listing 2.75 – Example of initializing a structure in Flash

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 Setup(void)
{
  path *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 = {5, 6, 10, 12, 255};
  ppt->direction = -1;
  (*ppt).direction = -1;
}
```

Listing 2.76 – 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` cannot be evaluated.

As an another example of pointer access consider the finite state machine controller for the `fsm[3]` structure shown previously. The state machine is illustrated below, along with the program.

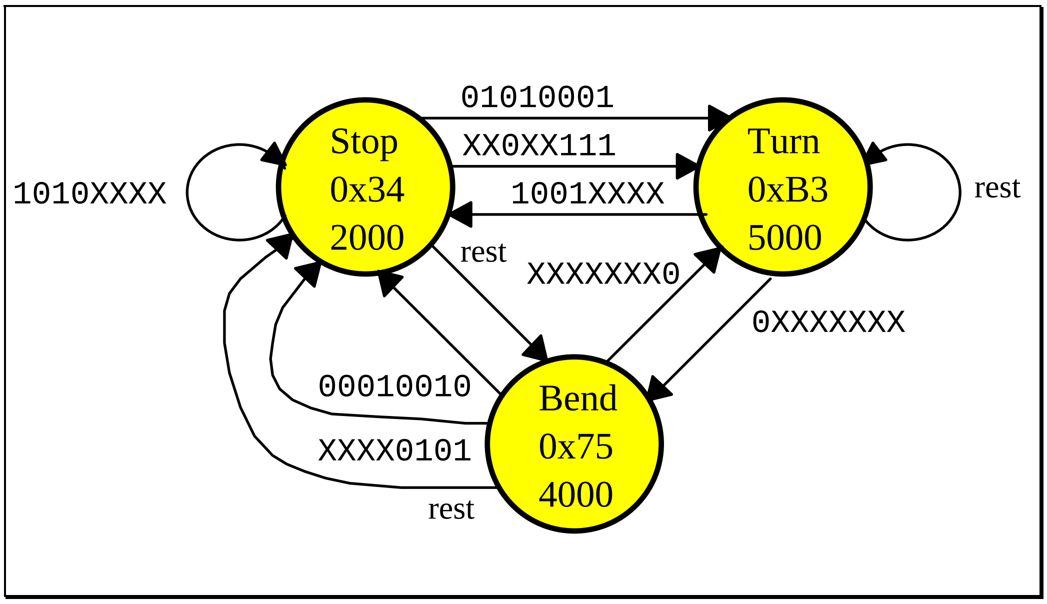


Figure 2.38 – Finite state machine

```
void control(void)
{
  PState pt;
  unsigned char input;
  unsigned short startTime;
  unsigned int i;

  TSCR1 |= 0x80;    // TEN(enable)
  TSCR2 = 0x01;     // timer/2 (500ns)
  DDRA = 0xFF;      // PortA bits 7-0 are outputs
  DDRB = 0x00;      // PortB bits 7-0 are inputs
  pt = stop;        // Initial State

  while(1)
  {
    // 1) output
    PORTA = pt->out;
    // Time (500 ns each) to wait
    startTime = TCNT;
    // 2) wait
    while ((TCNT - startTime) <= pt->wait);
    // 3) input
    input = PORTB;
    for (i = 0; i < 4; i++)
      if ((input & pt->andMask[i]) == pt->equMask[i])
      {
        // 4) next depends on input
        pt = pt->next[i];
        i = 4;
      }
  }
}
```

Listing 2.77 – Finite state machine controller for MC9S12

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 6-byte structure is copied on the stack when the function is called:

```c
unsigned char Input(port thePort)
{
  return (*thePort.addr);
}
```

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

```c
typedef const struct
{
  int mode;  // 0 for I/O, 1 for in only -1 for out only
  unsigned char volatile *addr; // address
  unsigned char volatile *ddr;  // direction reg
} port;

port PortJ =
{
  0,
  (unsigned char volatile *)(0x0028),
  (unsigned char volatile *)(0x0029)
};

int MakeOutput(port *ppt)
{
  if (ppt->mode == 1)
    return 0; // input only
  if (ppt->mode == -1)
    return 1; // OK, output only
  (*ppt->ddr) = 0xff; // make output
  return 1;
}

int MakeInput(port *ppt)
{
  if (ppt->mode == -1)
    return 0; // output only
  if (ppt->mode == 1)
    return 1;  // OK, input only
  (*ppt->ddr) = 0x00; // make input
  return 1;
}

unsigned char Input(port *ppt)
{
  return *ppt->addr;
}
```

```
void Output(port *ppt, unsigned char data)
{
  *ppt->addr = data;
}

void main(void)
{
  unsigned char myData;

  MakeInput(&PortJ);
  MakeOutput(&PortJ);
  Output(&PortJ, 0);
  myData = Input(&PortJ);
}
```

Listing 2.78 – 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 one-dimensional 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:

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

TNode *HeadPt;
```

Listing 2.79 – Linear linked list node structure

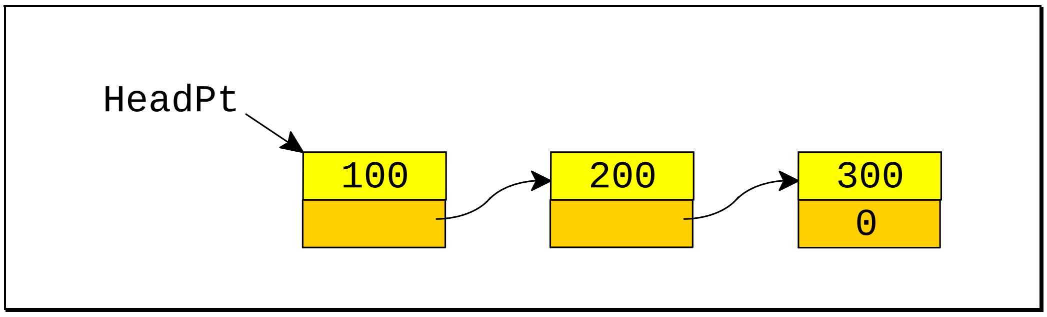


Figure 2.39 – Linear linked list with 3 nodes

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)
{
  TNode *pt;
  pt = malloc(sizeof(TNode));  // 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 2.80 – 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.

```
TNode *Search(unsigned short info)
{
  TNode *pt;

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

Listing 2.81 – 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)
{
  TNode *pt;
  unsigned short count;

  count = 0;
  pt = HeadPt;
  while (pt)
  {
    cnt++;
    pt = pt->next;    // link to next
  }
  return (count);
}
```

Listing 2.82 – 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.

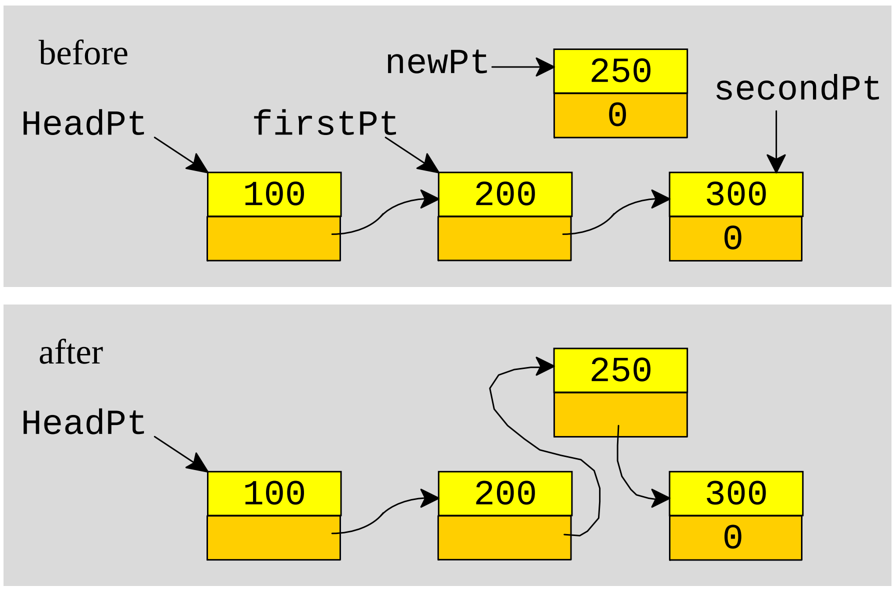


Figure 2.40 – Inserting a node in sorted order

There are 4 cases to consider. 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 previous 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.

```
#include <stdlib.h>;

int InsertData(unsigned short info)
{
  TNode *firstPt, *secondPt, *newPt;

  newPt = malloc(sizeof(TNode));  // create a new entry
  if (newPt)
  {
    newPt->data = info;       // store data

    // case 1
    if (HeadPt == 0)
    {
      newPt->next = HeadPt;   // only element
      HeadPt = newPt;
      return 1;
    }

    // case 2
    if (info <= HeadPt->data)
    {
      newPt->next = HeadPt;   // first element in list
      HeadPt = newPt;
      return 1;
    }

    // case 3
    firstPt = HeadPt;         // search from beginning
    secondPt = HeadPt->next;
    while (secondPt)
    {
      if (info <= secondPt->data)
      {
        newPt->next = secondPt;   // insert element here
        firstPt->next = newPt;
        return 1;
      }
      firstPt = secondPt;     // search next
      secondPt = secondPt->next;
    }

    // case 4
    newPt->next = secondPt;   // insert at end
    firstPt->next = newPt;
    return 1;
  }
  return 0;                  // out of memory
}
```

Listing 2.83 – 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 originally had only one element. This is okay 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.

```
#include <stdlib.h>;

int Remove(unsigned short info)
{
  TNode *firstPt, *secondPt;

  // case 1
  if (HeadPt == 0)
    return 0;   // empty list

  // case 2
  firstPt = HeadPt;
  secondPt = HeadPt->next;
  if (info == HeadPt->data)
  {
    HeadPt = secondPt; // remove first element in list
    free(firstPt);     // return unneeded memory to heap
    return 1;
  }

  // case 3
  while (secondPt)
  {
    if (secondPt->data == info)
    {
      firstPt->next = secondPt->next; // remove this one
      free(secondPt);   // return unneeded memory to heap
      return 1;
    }
    firstPt = secondPt;   // search next
    secondPt = secondPt->next;
  }

  // case 4
  return 0;    // not found
}
```

Listing 2.84 – 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.

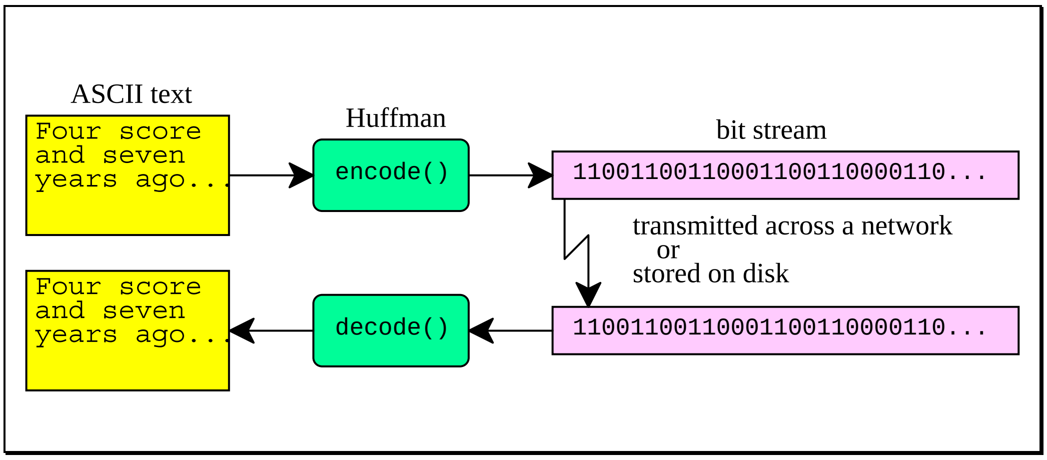

ASCII text
Four score and seven years ago...
Huffman
encode()
bit stream
11001100110001100110000110...
transmitted across a network or stored on disk
11001100110001100110000110...
decode()
Four score and seven years ago...

Figure 2.41 – 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 t a o i n" (that have a higher probability of occurrence) and long codes for seldom used consonants like "j x q 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 and 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.

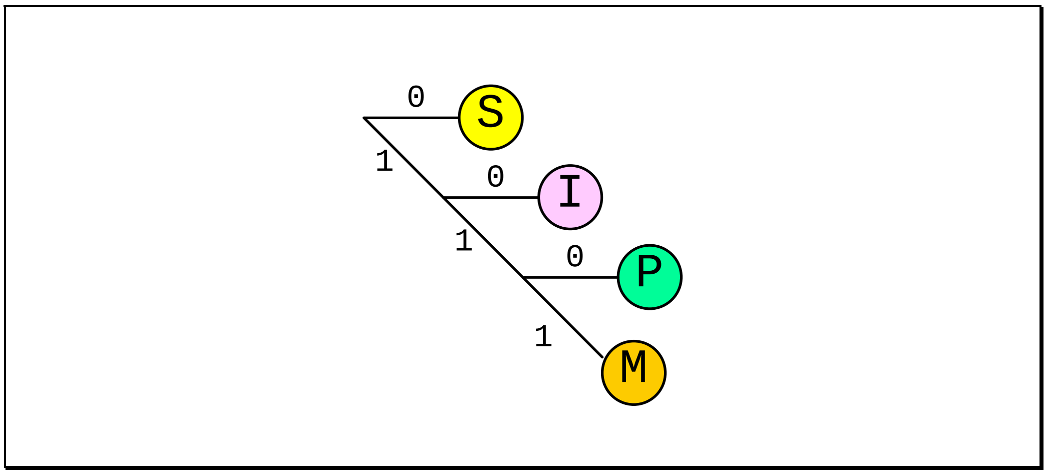


Figure 2.42 – Huffman code for the letters S I P M

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.

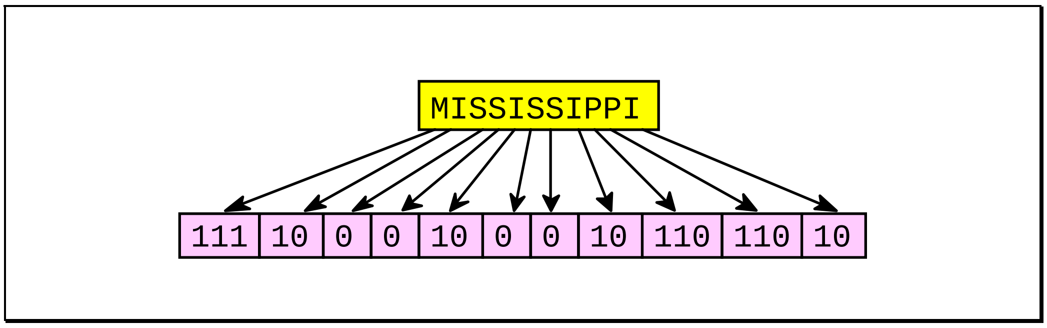


Figure 2.43 – Huffman encoding for MISSISSIPPI

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 saving. Nevertheless, for information that has a wide range of individual probabilities of occurrence, a Huffman code will be efficient.

2.178

In the following implementation the functions `BitPut()` and `BitGet()` are called to save and recover binary data. The implementations of these two functions are not shown.

```
const 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 TNode;
typedef TNode * PNode;

// Huffman tree
TNode twentysixth= {'Q','Z',0};
TNode twentyfifth= {'X',0,&twentysixth};
TNode twentyfourth={'J',0,&twentyfifth};
TNode twentythird= {'K',0,&twentyfourth};
TNode twentysecond={'V',0,&twentythird};
TNode twentyfirst= {'B',0,&twentysecond};
TNode twentyth=    {'P',0,&twentyfirst};
TNode ninteenth=   {'Y',0,&twentyth};
TNode eighteenth=  {'G',0,&ninteenth};
TNode seventeenth= {'F',0,&eighteenth};
TNode sixteenth=   {'W',0,&seventeenth};
TNode fifteenth=   {'M',0,&sixteenth};
TNode fourteenth=  {'C',0,&fifteenth};
TNode thirteenth=  {'U',0,&fourteenth};
TNode twelfth=     {'L',0,&thirteenth};
TNode eleventh=    {'D',0,&twelfth};
TNode tenth=       {'R',0,&eleventh};
TNode ninth=       {'H',0,&tenth};
TNode eighth=      {'S',0,&ninth};
TNode seventh=     {' ',0,&eighth};
TNode sixth=       {'N',0,&seventh};
TNode fifth=       {'I',0,&sixth};
TNode fourth=      {'O',0,&fifth};
TNode third=       {'A',0,&fourth};
TNode second=      {'T',0,&third};
TNode root=        {'E',0,&second};
```

```c
// ********encode***************
// convert ASCII string to Huffman bit sequence
// input is a null-terminated ASCII string
// returns bit count if OK
// returns 0          if BitFifo full
// returns 0xFFFF     if illegal character

int encode(char *sPt)
{
  int notFound;
  char data;
  int bitCount = 0;       // number of bits created
  PNode 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
// output is a null-terminated ASCII string
// will remove from the BitFifo until it is empty
// returns character count

int decode(char *sPt)
{
  int charCount = 0;    // number of ASCII characters created
  int notFound;
  unsigned int data;
  PNode 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 2.85 – A Huffman code implementation

Functions

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 into modules. A module is a software task that takes inputs and 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.

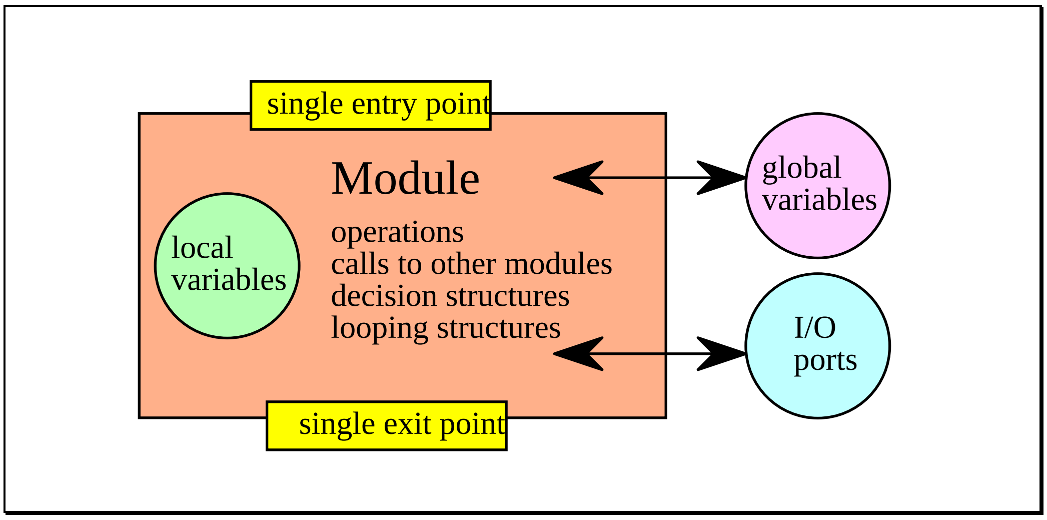


Figure 2.44 – A module has inputs and outputs

As a programmer we must take special care 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.

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:

```
// T+2 degrees Fahrenheit plus 4 degrees Centigrade
FtoC(T + 2) + 4;
```

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, 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 a function declaration as a prototype. Since the C compiler is essentially 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 Init(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 16-bit      16-bit
void OutString(char*);           // pointer to 8-bit none
char *alloc(int);                // 16-bit          pointer to 8-bit
int Exec(void(*fnctPt)(void));  // function pointer 16-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
void OutChar(char);   // Output a character
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, alternative declarations of these same five functions are given below:

```
    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 powerful features of C is to define pointers to functions. A simple example follows:

```
// pointer to a function with input and output
int (*fp)(int);

int fun1(int input)
{
  return(input + 1);    // this adds 1
}

int fun2(int input)
{
  return(input + 2);    // this adds 2
}

void Setup(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 2.86 – 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 takes one integer argument and returns an integer. 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 16-bit signed input parameter that returns a 16-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. 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 2.86.

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 projects 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 COP functions, and some 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 "COP.h"
#include "Timer.h"

void main(void)
{
  char letter;
  short n = 0;

  COP_Init();
  LCD_Init();
```

```
    Timer_Init()
    LCD_String("This is a LCD");
    Timer_MsWait(1000);
    LCD_Clear();
    letter = 'a' - 1;
    while(1)
    {
      if (letter == 'z')
        letter = 'a';
      else
        letter++;
      LCD_PutChar(letter);
      Timer_MsWait(250);
      if (++n == 16)
      {
        n = 0;
        LCD_Clear();
      }
    }
  }
```

Listing 2.87 – Modular approach to software development

C function definitions have the following form:

```
type Name(parameter list)
{
  Compound Statement
};
```

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. CodeWarrior passes the input parameters from left to right on the stack. If the last parameter has a simple type, it is not pushed but passed in a register. Function results are returned in registers, except if the function returns a result larger than 32 bits. Functions returning a result larger than 32 bits are called with an additional parameter. This parameter is the address where the result should get copied.

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, 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 *`Compound Statement`*. 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 a function named `add3` which takes three input arguments:

```
int add3(int z1, int z2, int z3)
{
  int y;
  y = z1 + z2 + z3;
  return(y);
}
```

Listing 2.88 – Example function with 3 inputs and one output

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. 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. CodeWarrior allocates the stack space for the parameters at the start of the code that will make the function call. Then the values are stored into the pre-allocated stack position before it calls the function. The input parameters are removed from the stack at the end of the function. The return parameter is generally located in a register.

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.

Function calls can appear in expressions. 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);
  return (data);
}

void main(void)
{
  int result;

  result = execute(&fun1); // result = fun1(5);
  result = execute(&fun2); // result = fun2(5);
}
```

Listing 2.89 – 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

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 sometimes 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. The function `FIFO_Get`, shown below, returns two parameters. The return parameter is an integer 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 `FIFO_Get` will store the return parameter into a local variable of `InChar`. Normally `FIFO_Get` 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 `FIFO_Get`.

```c
int FIFO_Get(char *datapt)
{
  if (Size == 0 )
    return(0);              // Empty if Size == 0
  *datapt = FIFO[GetI++];   // Get data out of FIFO
  Size--;
  if (GetI == FIFO_SIZE)
    GetI = 0;               // Wrap
  return(-1);               // Successful
}

char InChar(void)
{
  char data;
  while (FIFO_Get(&data));
  return (data);
}
```

Listing 2.90 – Multiple output parameters 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 `FIFO_Put()` of the example below, there will be three separate and distinct copies of the 0x41 data (`main`, `OutChar` and `FIFO_Put`).

```
int FIFO_Put (char data)
{
  if (Size == FIFO_SIZE)
    return(0);                // Failed, FIFO was full

  Size++;
  FIFO[PutI++] = data;        // Put data into FIFO
  if (PutI == FIFO_SIZE)
    PutI = 0;                 // Wrap
  return(-1);                 // Successful
}

void OutChar(char data)
{
  while (PutFifo(data));
  SC0CR2 = 0xAC;
}

void main(void)
{
  char data = 0x41;

  OutChar(data);
}
```

Listing 2.91 – 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 effect whatsoever on the objects that might have supplied their values. They can be changed at will and their sources will not be affected in any way. This removes a burden of concern for a programmer since they 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. Since any expression is a valid argument, addresses can be passed to functions.

Since expressions may include assignment, increment, and decrement operators, 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)`.

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 ANSI C library. To write a function with a variable number of arguments, you need to consult a reference on advanced C programming.

Private versus Public Functions

For every function definition, CodeWarrior 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 2.87 shown earlier. Let's 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 Timer_Init(void);
void Timer_MsWait(unsigned int time);
```

Listing 2.92 – Timer.h header file has public functions

The implementations of all functions are written in the `Timer.c` file. The function, `TimerWait`, is private and can only be called by software inside the `Timer.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 cannot be accessed by software outside the `Timer.c` file.

```
static unsigned short TimerClock; // private global

// public function
void Timer_Init(void)
{
  TSCR1 |= 0x80;      // TEN(enable)
  TSCR2 = 0x01;       // timer/2 (500ns)
  TimerClock = 2000; // 2000 counts per ms
}

// private function
static void TimerWait(unsigned short time)
{
  TC5 = TCNT + TimerClock;  // 1.00ms wait
  TFLG1 = 0x20;             // clear C5F
  while ((TFLG1&0x20) == 0);
}

// public function
void Timer_MsWait(unsigned short time)
{
  for (; time > 0; time--)
    TimerWait(TimerClock); // 1.00ms wait
}
```

Listing 2.93 – Timer.c implementation file defines all functions

Finite State Machine using Function Pointers

Now that we have seen how to declare, initialize and access function pointers, we can create very flexible finite state machines. In the finite state machine presented in Listing 2.75 and Listing 2.77, the output was a simple number that is written to the output port. In the next example, we will implement the exact same FSM, 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 FSM.

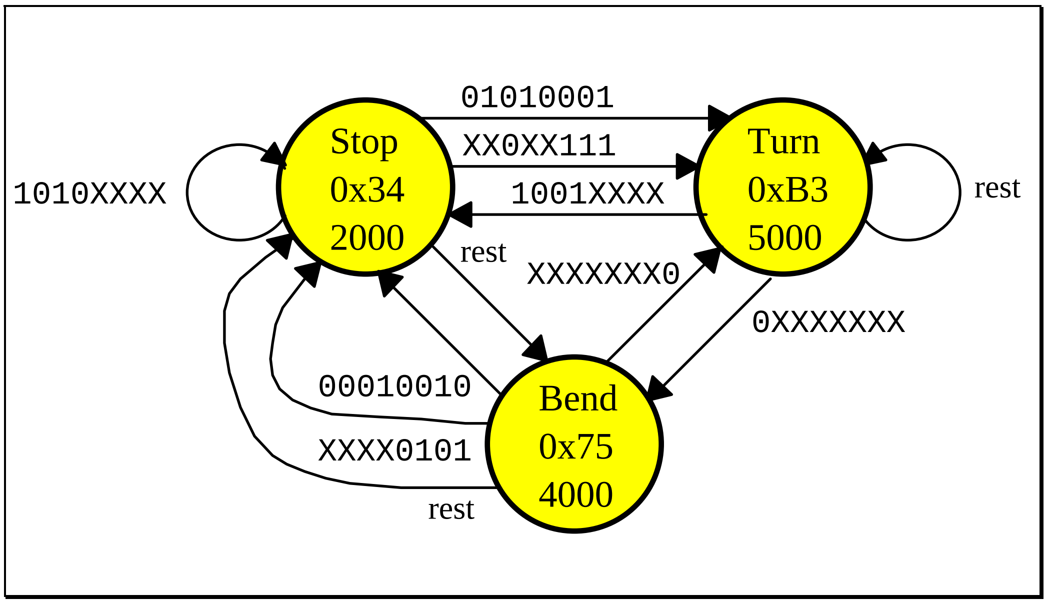


Figure 2.45 – Finite state machine

Compare the following implementation to Listing 2.75, and see that the **unsigned char** Out; constant is replaced with a **void** (*CmdPt)(**void**); function pointer. The three general functions DoStop(), DoTurn(), and DoBend() are also added.

```
typedef const struct State
{
  void (*cmdPt)(void);          // function to execute
  unsigned short wait;          // Time (bus cycles) to wait
  unsigned char andMask[4];
  unsigned char equMask[4];
  const struct State *next[4]; // Next states
} TState;

typedef TState *PState;

#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;
}

TState 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 2.94 – Linked finite state machine structure stored in Flash

Compare the following implementation to Listing 2.77, and see that the `PORTA = 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;

  TSCR1 |= 0x80;    // TEN(enable)
  TSCR2 = 0x01;     // timer/2 (500ns)
  DDRA = 0xFF;      // PortA bits 7-0 are outputs
  DDRB = 0x00;      // PortB bits 7-0 are inputs
  pt = stop;        // Initial State

  while(1)
  {
    // 1) execute function
    (*pt->cmdPt)();
    // Time (500 ns each) to wait
    startTime = TCNT;
    // 2) wait
    while ((TCNT - startTime) <= pt->wait);
    // 3) input
    input = PORTB;
    for (i = 0; i < 4; i++)
      if ((input & pt->andMask[i]) == pt->equMask[i])
      {
        // 4) next depends on input
        pt = pt->next[i];
        i = 4;
      }
  }
};
```

Listing 2.95 – Finite state machine controller for MC9S12

Linked List Interpreter using Function Pointers

In the next example, function pointers are stored in a linked 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
typedef const struct Node
{
  unsigned char letter;
  void (*fnctPt)(void);
  const struct Node *next;
} TNode;

typedef TNode *NodePtr;

void CommandA(void)
{
  OutString("\nExecuting Command a");
}

void CommandB(void)
{
  OutString("\nExecuting Command b");
}

void CommandC(void)
{
  OutString("\nExecuting Command c");
}

TNode LL[3] =
{
  {'a', &CommandA, &LL[1]},
  {'b', &CommandB, &LL[2]},
  {'c', &CommandC, NULL}
};
```

```
void main(void)
{
  NodePtr pt;
  char string[40];

  SCI_Init();     // Enable SCI port
  TSCR |=0x80;    // TEN(enable)
  TMSK2 = 0xA2;   // TOI arm, TPU(pullup) timer/4 (500ns)
  OutString("\nEnter a single letter followed by <enter>");
  while (1)
  {
    OutString("\n>");
    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)
          OutString(" Error");
      }
    }
  }
}
```

Listing 2.96 – Linked list implementation of an interpreter

Compare the syntax of the function call, `(*pt->cmdPt)();`, in Listing 2.95, with the syntax in this example, `pt->fnctPt();`. In the CodeWarrior compiler, these two expressions both generate code that executes the function.

Preprocessor Directives

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
- conditional compiling
- inclusion of text from other files
- implementation-dependent features

The preprocessor is controlled by directives which are not part of the C language. 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. CodeWarrior 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.

Macros define names which stand for arbitrary strings of text:

```
#define Name CharacterString
```

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, it is simply a literal replacement of one set of characters with another.

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 earlier. `CharacterString` begins with the first printable character following `Name` and continues through to 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*. The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

However, we may replace a name with anything at all: a commonly occurring expression or sequence of statements for instance. To disable interrupts during a critical section we could implement:

```
#define START_CRITICAL { asm tfr ccr,a; asm staa savedCCR; asm sei; }
#define END_CRITICAL   { asm ldaa savedCCR; asm tfr a,ccr; }

             void function(void)
             {
               unsigned char savedCCR;
               ...
               START_CRITICAL; // make atomic, entering critical section
               // we have exclusive access to global variables
               ...
               END_CRITICAL; // end critical section
             }
```

Listing 2.97 – Example of `#define`

There is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code (but if it does not, you may get error messages from the C compiler when you use the macro).

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 2.98 – 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 code. In this example the only purpose of writing to PORTC is to assist in performance debugging. Once the system is debugged, we can remove all the debugging code, simply by deleting the `#define Debug` line.

```
#define Debug

int Sub(int j)
{
  int i;

#ifdef Debug
  PORTC |= 0x01;  // PC0 set when Sub is entered
#endif
  i = j + 1;
#ifdef Debug
  PORTC &= ~0x01; // PC0 cleared when Sub is exited
#endif
  return(i);
}
```

```
void ProgA(void)
{
  int i;

#ifdef Debug
  PORTC |= 0x02; // PC1 set when ProgA is entered
#endif
  i = Sub(5);
  while (1)
  {
    i = Sub(i);
  }
}

void ProgB(void)
{
  int i;

  i = 6;
  ...
#ifdef Debug
  PORTC &= ~0x02; // PC1 cleared when ProgA is exited
#endif
}
```

Listing 2.99 – Conditional compilation can help in 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 PC file specification format, including drive, path, filename, and extension.

Implementation-Dependent Features

The `#pragma` directive is used to instruct the compiler to use pragmatic or implementation-dependent features. For example, in CodeWarrior you can declare an interrupt service routine using the `#pragma TRAP_PROC` directive:

```
#pragma TRAP_PROC
void MyInterruptServiceRoutine(void)
{
  ...
}
```

The CodeWarrior compiler will then use the `rti` instruction rather than the `rts` instruction to return from the function. However, CodeWarrior was designed for embedded systems, and defines a non-ANSI C keyword **`interrupt`**. We can use the **`interrupt`** key word to specify a function as an interrupt handler, as well as specify its location in the interrupt vector table:

```
void interrupt 23 KeyWakeUpPortJ(void)
{
  KWIFJ = 0x80; // clear flag
  FIFO_Put(PORTJ & 0x7F);
}
```

Listing 2.100 – Interrupt service routines as specified in CodeWarrior

Assembly Language Programming

One of the main reasons for using the C language is to achieve portability. But there are occasional situations in which it is necessary to sacrifice portability in order to gain full access to the operating system or to the hardware in order to perform some interface requirement. If these instances are kept to a minimum and are not replicated in many different programs, the negative effect on portability may be acceptable. There are two approaches to writing assembly language with CodeWarrior. The first method inserts a single assembly instruction directly into a C function using the `asm string;` feature. Everything within the `string` statement is assumed to be assembly language code and is sent straight to the output of the compiler exactly as it appears in the input. The second approach is to write an entire file in assembly language, which may include global variables and functions. In CodeWarrior, we include assembly files by adding them to the project. Entire assembly files can also be assembled separately then linked at a later time to the rest of the program. The simple insertion method is discussed in this section.

How to Insert Single Assembly Instructions

To support this capability, C provides for assembly language instructions to be written into C programs anywhere a statement is valid. Since the compiler generates assembly language as output, when it encounters assembly language instructions in the input, it simply copies them directly to the output.

A special directive delimits assembly language code. The following example inserts the assembly language instruction `cli` (enable interrupts) into the program at that point.

```
asm cli
```

A better way is to #define macros:

```
#define INTR_ON()  asm cli
#define INTR_OFF() asm sei
```

The following function runs with interrupts disabled.

```
void FIFO_Init(void)
{
  INTR_OFF();      // make atomic, entering critical section
  PutI=GetI=Size=0;// Empty when Size == 0
  INTR_ON();       // end critical section
}
```

Listing 2.101 – Example of an assembly language macro

Of course, to make use of this feature, we must know how the compiler uses the CPU registers, how functions are called, and how the operating system and hardware works. It will certainly cause a programming error if your embedded assembly modifies the stack pointer, SP, or the stack frame pointer, X. On the other hand, in most situations you should be able to modify the CCR, A, B, or Y without causing a program error. It is good practice to observe the resulting assembly output of the entire function to guarantee that the embedded assembly has not affected the surrounding C code. Unfortunately, this verification must be repeated when you change or upgrade the compiler.

In CodeWarrior you can access a global or local variable directly using just its name, and the compiler will convert it to the appropriate addressing mode.

```
short time;

void Add1time(void)
{
  asm ldy time
  asm iny
  asm sty time
}
```

Listing 2.102 – Example of an assembly language access to a global variable