Embedded Software

Lecture Notes

2014
# Contents

**LECTURE 1 – EMBEDDED SYSTEMS**

- Overview ............................................................................................................. 1.1
- Embedded Systems Characteristics ................................................................. 1.3
- The UTS ModCon board ................................................................................... 1.4

**LECTURE 2 – EMBEDDED C**

- Program Structure .......................................................................................... 2.1
  - Case Study 1: Microcomputer-based lock ................................................. 2.1
  - Case Study 2: A Serial Port MC9S12 Program ...................................... 2.4
- Free Field Language ..................................................................................... 2.6
- Precedence .................................................................................................. 2.12
- Associativity .............................................................................................. 2.13
- Comments ..................................................................................................... 2.14
- Preprocessor directives .............................................................................. 2.15
- Global declarations ................................................................................... 2.16
- Declarations and Definitions .................................................................. 2.16
- Functions ..................................................................................................... 2.18
  - Compound Statements ........................................................................ 2.21
  - Global variables .................................................................................... 2.23
  - Local variables ....................................................................................... 2.24
  - Source Files ............................................................................................ 2.25
- Tokens ............................................................................................................... 2.28
  - ASCII character Set ............................................................................ 2.29
  - Literals .................................................................................................... 2.30
  - Keywords .................................................................................................. 2.31
  - Names ........................................................................................................... 2.32
  - Punctuation ................................................................................................ 2.35
    - Semicolons .......................................................................................... 2.35
    - Colons .................................................................................................. 2.36
  - Commas......................................................................................................... 2.37
  - Apostrophes ............................................................................................ 2.38
  - Quotation marks .................................................................................... 2.38
  - Braces ........................................................................................................ 2.38
  - Brackets ...................................................................................................... 2.39
  - Parentheses ............................................................................................... 2.39
- Operators .................................................................................................... 2.40
  - Numbers, characters and strings ............................................................... 2.41
    - Binary representation ........................................................................ 2.41
    - 8-bit unsigned numbers ..................................................................... 2.43
    - 8-bit signed numbers ......................................................................... 2.46
    - 16 bit unsigned numbers .................................................................. 2.49
    - 16-bit signed numbers ....................................................................... 2.50
  - *typedefs* for signed and unsigned data types ..................................... 2.51
- Big- and little-endian .................................................................................... 2.52
- Boolean information ................................................................................... 2.54
- Decimal numbers ........................................................................................ 2.54
- Octal numbers ............................................................................................. 2.57
ARRAY DECLARATIONS .................................................. 2.153
ARRAY REFERENCES ..................................................... 2.154
POINTERS AND ARRAY NAMES ..................................... 2.154
NEGATIVE SUBSCRIPTS ............................................... 2.155
ADDRESS ARITHMETIC ................................................. 2.156
STRING FUNCTIONS IN STRING.H ................................. 2.157
A FIFO QUEUE EXAMPLE USING INDICES ...................... 2.162
STRUCTURES .............................................................. 2.164
STRUCTURE DECLARATIONS ........................................ 2.164
ACCESSING MEMBERS OF A STRUCTURE ....................... 2.166
INITIALIZATION OF A STRUCTURE ............................... 2.167
USING POINTERS TO ACCESS STRUCTURES .................. 2.168
PASSING STRUCTURES TO FUNCTIONS ......................... 2.170
LINEAR LINKED LISTS .................................................. 2.171
EXAMPLE OF A HUFFMAN CODE ................................. 2.176
FUNCTIONS ................................................................. 2.181
FUNCTION DECLARATIONS ........................................... 2.183
FUNCTION DEFINITIONS ............................................... 2.186
FUNCTION CALLS ......................................................... 2.189
ARGUMENT PASSING .................................................. 2.191
PRIVATE VERSUS PUBLIC FUNCTIONS ........................... 2.194
FINITE STATE MACHINE USING FUNCTION POINTERS .... 2.195
LINKED LIST INTERPRETER USING FUNCTION POINTERS .... 2.198
PREPROCESSOR DIRECTIVES ........................................ 2.200
MACRO PROCESSING .................................................... 2.200
CONDITIONAL COMPILING .......................................... 2.203
INCLUDING OTHER SOURCE FILES .............................. 2.205
IMPLEMENTATION-DEPENDENT FEATURES .................... 2.206
ASSEMBLY LANGUAGE PROGRAMMING .......................... 2.207
HOW TO INSERT SINGLE ASSEMBLY INSTRUCTIONS ....... 2.207

LECTURE 5 – INTERRUPTS
INTRODUCTION ............................................................. 5.1
INTERRUPTS ................................................................. 5.2
USING INTERRUPTS ..................................................... 5.3
INTERRUPT PROCESSING ............................................... 5.4
INTERRUPT POLLING .................................................... 5.4
interrupt service routines (isrs) ................................. 5.5
Declaring interrupt service routines in C ....................... 5.6
Enabling and Disabling Interrupts ............................... 5.6
Interrupt Latency ......................................................... 5.6
Exceptions ................................................................. 5.9
Foreground / Background Threads .............................. 5.10
Serial Communication Interface using Interrupts ............. 5.11
Output Device Interrupt Request on Transition to Ready ..... 5.12
Output Device Interrupt Request on Ready .................... 5.14
Communicating between Threads ............................... 5.15
Critical Sections in C .................................................... 5.16
Interrupt Priority ........................................................ 5.18
LECTURE 6 – TIMING GENERATION & MEASUREMENT

INTRODUCTION ...................................................................................................... 6.1
TIMER MODULE ..................................................................................................... 6.1
MODULUS DOWN-COUNTER .................................................................................. 6.2
OUTPUT COMPARE .............................................................................................. 6.3
COUPLED OUTPUT COMPARE ............................................................................ 6.5
INPUT CAPTURE ..................................................................................................... 6.5
PULSE ACCUMULATOR ........................................................................................ 6.7
FURTHER INFORMATION ...................................................................................... 6.7

LECTURE 7 – THE EMBEDDED SOFTWARE TOOL CHAIN

INTRODUCTION ...................................................................................................... 7.1
OVERVIEW ............................................................................................................. 7.1
PREPROCESSOR ................................................................................................. 7.1
COMPILED ......................................................................................................... 7.2
ASSEMBLER ...................................................................................................... 7.2
LINKER ............................................................................................................. 7.2
LOADER ............................................................................................................ 7.2
BUILDING A SOFTWARE PROJECT ........................................................................ 7.3
THE INTEGRATED DEVELOPMENT ENVIRONMENT ................................................. 7.4
THE PROJECT MANAGER .................................................................................. 7.4
BUILD TARGETS .............................................................................................. 7.5
THE EDITOR ........................................................................................................... 7.6
EDITING ............................................................................................................ 7.6
NAVIGATING ..................................................................................................... 7.6
THE COMPILER AND ASSEMBLER ..................................................................... 7.7
THE LINKER ......................................................................................................... 7.8
THE LOADER ..................................................................................................... 7.9
THE DEBUGGER ................................................................................................ 7.9
THE BURNER ..................................................................................................... 7.9
THE BOOTLOADER ............................................................................................ 7.9

LECTURE 8 – CONCURRENT SOFTWARE

INTRODUCTION ...................................................................................................... 8.1
THREADS ......................................................

THREAD CONTROL BLOCKS (TCBs) ................................................................. 8.6
SCHEDULERS ......................................................................................................... 8.7
OTHER SCHEDULING ALGORITHMS ................................................................. 8.9
TIMER OVERVIEW .............................................................................................. 8.9
OPERATING SYSTEMS .......................................................................................... 8.13
THE SEMAPHORE ........................................................................................... 8.13
MUTUAL EXCLUSION WITH SEMAPHORES ......................................................... 8.14
SYNCHRONISATION USING SEMAPHORES ......................................................... 8.16
THE PRODUCER / CONSUMER PROBLEM USING SEMAPHORES ...................... 8.17
Lecture 1 – Embedded Systems

Overview. Embedded systems characteristics. The UTS ModCon board.

Overview

Computing systems are everywhere. Billions of computing systems are built every year that are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device’s user.

A quick look around our environment turns up embedded systems in a surprising number of places. The picture below shows just a few such systems in common environments.

Figure 1.1
A listing of these systems is given below:

### Outdoors

1. Helicopter: control, navigation, communication, etc.
2. Medicine administering systems
3. Smart hospital bed with sensors and communication
4. Patient monitoring system
5. Surgical displays
6. Ventilator
7. Digital thermometer
8. Portable data entry systems
9. Pacemaker
10. Automatic door
11. Electric wheelchair
12. Smart briefcase with fingerprint enabled lock
13. Ambulance: medical and communication equipment
14. Automatic irrigation systems
15. Jet aircraft: control, navigation, communication, autopilot, collision-avoidance, in-flight entertainment, passenger telephones, etc.
16. Laptop computer (contains embedded systems)
17. Mobile telephone
18. Portable stereo
19. Satellite receiver system
20. Credit / debit card reader
21. Barcode scanner
22. Cash register
23. Automatic teller machine
24. Car (engine control, cruise control, temperature control, music system, anti-lock brakes, active suspension, navigation, toll transponder, etc.)
25. Automatic lighting
26. Pump monitoring system
27. Lottery ticket dispenser
28. Pager
29. Traffic light controller

### Indoors

34. Cordless phone
35. Coffee maker
36. Rice cooker
37. Portable radio
38. Programmable oven
39. Microwave oven
40. Smart refrigerator
41. In-home computer network router
42. Clothes dryer
43. Clothes washing machine
44. Portable MP3 player
45. Digital camera
46. Electronic book
47. Garbage compactor
48. Hearing aid
49. Dishwasher
50. Electronic clock
51. Video camera
52. Electronic wristwatch
53. Pager
54. Mobile phone
55. CD player
56. DVD player
57. Smart speakers
58. Stereo receiver
59. TV set-top box
60. Television
61. PVR
62. TV-based Web access box
63. House temperature control
64. Home alarm system
65. Point-of-sale system
66. Video-game console
67. TV remote control
68. Electronic keyboard
69. Fax machine
70. Scanner
30. Police car (data lookup, communication, sirens, radar, etc.)
31. Mobile phone base station
32. Hand-held communicator (walkie-talkie)
33. Fire-control onboard computer
71. Wireless networking
72. Telephone modem
73. ADSL modem
74. Printer
75. Portable video game
76. Personal digital assistant
77. Portable digital picture viewer
78. Phone with answering machine

Nearly any device that runs on electricity either already has or soon will have a computing system embedded within it. In 2011, 500 million smart phones, 60 million tablet PCs and 25 million eReaders were shipped.¹

**Embedded Systems Characteristics**

Embedded systems have several common characteristics that distinguish such systems from other computing systems:

1. *Single-functioned:* An embedded system usually executes a specific program repeatedly.

2. *Tightly constrained:* Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real time, must consume minimum power to extend battery life, and must be designed rapidly to capture market windows.

3. *Reactive and real-time:* Many embedded systems must continually react to changes in the system’s environment and must compute certain results in real time without delay.

---

The UTS ModCon Board

The UTS Modular Controller (ModCon) board is based on the Freescale MC9S12 16-bit microcontroller. It has various optionally-mounted peripherals, expansion ports, and digital and analog interfaces that are designed to be interfaced to other hardware in a modular fashion. It also has a USB interface to enable communication with a PC.

According to Freescale, “The S12 MCU family is the most widely adopted 16-bit architecture in the automotive market. Freescale has shipped approximately 400 million 16-bit S12 and S12X MCUs”\(^2\) as of 5 November 2008, and continue to ship at a rate of more than 100 million units per year.

Lecture 2 – Embedded C


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.

![Digital lock hardware](image)

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.

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.

   \[
   \texttt{#define PORTA} \ (*\texttt{(unsigned char volatile *)} \(0x0000\))
   \texttt{#define DDRA} \ (*\texttt{(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.

   \[
   \texttt{unsigned int} \ \texttt{Count;}
   \]

   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.

   \[
   \texttt{const unsigned char} \ \texttt{KEY} = 0x23; \ /* \texttt{key code} */
   \]

   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.

![Flowchart and corresponding pseudo-code and C code]

**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.
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.

```c
/* 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)
#define TDRE 0x80

void SCI_Init(void)
{
    /* 9600 baud, 16 MHz Xtal assumed */
    BAUD = 0x34;
    /* enable SCI, no interrupts */
    SCCR2 = 0x0C;
}

void SCI_Out(unsigned char Data)
{
    /* Wait for TDRE to be set */
    while ((SCSR & TDRE) == 0);
    /* then output */
    SCDR = Data;
}

void main(void)
{
    unsigned char Info;
    /* turn on SCI serial port */
    SCI_Init();
    /* 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 2.1 – 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

Embedded Software 2014
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 \texttt{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 \texttt{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 \texttt{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 \texttt{SCI\_Open} could have been written without any line breaks

\begin{verbatim}
void SCI_Open(void) {BAUD=0x30;SCCR2=0x0C;}
\end{verbatim}

\begin{tabular}{|c|}
\hline
\textbf{Use a programming style that is easy to read} \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
\textbf{Since we rarely make hardcopy printouts of our software, it is not necessary to minimize the number of line breaks.} \\
\hline
\end{tabular}

\begin{figure}
\begin{itemize}
\item[(2.1)]
\end{itemize}
\end{figure}
Similarly we could have added extra line breaks

```c
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.

<table>
<thead>
<tr>
<th>Punctuation</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>semicolon</td>
<td>End of statement</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
<td>Defines a label</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>Separates elements of a list</td>
</tr>
<tr>
<td>( )</td>
<td>parentheses</td>
<td>Start and end of a parameter list</td>
</tr>
<tr>
<td>{ }</td>
<td>braces</td>
<td>Start and stop of a compound statement</td>
</tr>
<tr>
<td>[ ]</td>
<td>brackets</td>
<td>Start and stop of a array index</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>quotation marks</td>
<td>Start and stop of a string</td>
</tr>
<tr>
<td>' '</td>
<td>apostrophes</td>
<td>Start and stop of a character constant</td>
</tr>
</tbody>
</table>

Table 2.1 – Special characters can be punctuation marks
The next table shows the single character operators.

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

Table 2.2 – Special characters can be operators
The next table shows the operators formed with multiple characters.

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

Table 2.3 – Multiple special characters also can be operators
The following section illustrates some of the common operators. We begin with the assignment operator.

```c
/* 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.

```c
/* 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.

```c
#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).

```c
#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.

```c
#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.

```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;`.

Embedded Software 2014
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.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>$()$ $[]$ $.$ $-&gt;$ $++$(postfix) $--$(postfix)</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>$++$(prefix) $-(prefix)$ $!$ $~$ $sizeof$(type) $+(unary)$ $-(unary)$ $&amp;(address)$ $*(dereference)$</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td>$*$ $/$ $%$ $+$ $-$ $&lt;&lt;$ $&gt;&gt;$ $&lt;=$ $&gt;=$ $==$ $!=$</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>$&amp;$ $^$ $</td>
<td>$ $&amp;&amp;$ $||$ $?$</td>
</tr>
<tr>
<td></td>
<td>$= += -= *= /= %= &lt;&lt;= &gt;&gt;=$ $</td>
<td>= &amp;= ^=</td>
</tr>
<tr>
<td>lowest</td>
<td>$,$</td>
<td>left to right</td>
</tr>
</tbody>
</table>

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)
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 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:

```
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.

```c
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.

```c
#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:

- 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

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

```c
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:

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

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)
{
    // 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);
```

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.
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);
}

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:

void main(void)
{
    short a, b;
    a = add(2000, 2000);   // call to add
    short add(short x, short y)
    {
        short z;
        z = x + y;              // z = 4000
        if ((x > 0) && (y > 0) && (z < 0))
            z = 32767;
        if ((x < 0) && (y < 0) && (z > 0))
            z = -32768;
        return (z);
    }
    // return 4000 from call
    b = 0
    while (1)
    {
        b = add(b, 1);         // call to add
    }
}

void main(void)
{
    short a, b;
    a = add(2000, 2000);   // call to add
    short add(short x, short y)
    {
        short z;
        z = x + y;              // z = 4000
        if ((x > 0) && (y > 0) && (z < 0))
            z = 32767;
        if ((x < 0) && (y < 0) && (z > 0))
            z = -32768;
        return (z);
    }
    // return 1 from call
}

while (1)

Functions use parameters to receive input values, and sometimes return a single value.
2.20

18     {                      // call to add
19       b = add(b, 1);       
1     
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);          // return 1 from call
10    }                  
18     }                  

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 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:

```c
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.

**Compound Statements**

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

```c
// 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

```c
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.

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.

```c
// 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 preferably 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.

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

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

```c
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.

```c
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.

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.

```
// ****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
```
2.26

// ****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

// ****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:

- 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.

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:

```c
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:

```c
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).

<table>
<thead>
<tr>
<th>Bits 0 to 3</th>
<th>ASCII character codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
</tr>
</tbody>
</table>

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 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 to 122 or $61 to $7A), and
- the special characters (all the rest).

**Literals**

*Numeric literals* consist of an uninterrupted sequence of digits delimited by white spaces or special characters (operators or punctuation). Although CodeWarrior does support floating-point numbers, but this document will not cover them. The use of floating-point numbers requires a substantial amount 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, '	' 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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asm</code></td>
<td>Insert assembly code.</td>
</tr>
<tr>
<td><code>auto</code></td>
<td>Specifies a variable as automatic (created on the stack).</td>
</tr>
<tr>
<td><code>break</code></td>
<td>Causes the program control structure to finish.</td>
</tr>
<tr>
<td><code>case</code></td>
<td>One possibility within a switch statement.</td>
</tr>
<tr>
<td><code>char</code></td>
<td>8-bit integer.</td>
</tr>
<tr>
<td><code>const</code></td>
<td>Defines a global parameter as a constant in Flash, and defines a local parameter as a fixed value.</td>
</tr>
<tr>
<td><code>continue</code></td>
<td>Causes the program to go to beginning of loop.</td>
</tr>
<tr>
<td><code>default</code></td>
<td>Used in switch statement for all other cases.</td>
</tr>
<tr>
<td><code>do</code></td>
<td>Used for creating program loops.</td>
</tr>
<tr>
<td><code>double</code></td>
<td>Specifies a variable as double precision floating point.</td>
</tr>
<tr>
<td><code>else</code></td>
<td>Alternative part of a conditional.</td>
</tr>
<tr>
<td><code>extern</code></td>
<td>Defined in another module.</td>
</tr>
<tr>
<td><code>float</code></td>
<td>Specifies a variable as single precision floating point.</td>
</tr>
<tr>
<td><code>for</code></td>
<td>Used for creating program loops.</td>
</tr>
<tr>
<td><code>goto</code></td>
<td>Causes program to jump to specified location.</td>
</tr>
<tr>
<td><code>if</code></td>
<td>Conditional control structure.</td>
</tr>
<tr>
<td><code>int</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>long</code></td>
<td>32-bit integer.</td>
</tr>
<tr>
<td><code>register</code></td>
<td>Specifies how to implement a local.</td>
</tr>
<tr>
<td><code>return</code></td>
<td>Leave function.</td>
</tr>
<tr>
<td><code>short</code></td>
<td>16-bit integer.</td>
</tr>
<tr>
<td><code>signed</code></td>
<td>Specifies variable as signed (default).</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>Built-in function returns the size of an object.</td>
</tr>
<tr>
<td><code>static</code></td>
<td>Stored permanently in memory, accessed locally.</td>
</tr>
<tr>
<td><code>struct</code></td>
<td>Used for creating data structures.</td>
</tr>
<tr>
<td><code>switch</code></td>
<td>Complex conditional control structure.</td>
</tr>
<tr>
<td><code>typedef</code></td>
<td>Used to create new data types.</td>
</tr>
<tr>
<td><code>unsigned</code></td>
<td>Always greater than or equal to zero.</td>
</tr>
<tr>
<td><code>void</code></td>
<td>Used in parameter list to mean no parameter.</td>
</tr>
<tr>
<td><code>volatile</code></td>
<td>Can change implicitly outside the direct action of the software. It disables compiler optimization, forcing the compiler to fetch a new value each time.</td>
</tr>
<tr>
<td><code>while</code></td>
<td>Used for creating program loops.</td>
</tr>
</tbody>
</table>

Table 2.6 – Keywords have predefined meanings
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

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`

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

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++.

```c
/* ********** 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];
```
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)
{
        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
    {
        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,

```c
#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

```c
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.

```c
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.

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

Listing 2.22 – 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,

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

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

```c
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);
}
```
```c
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`.

```c
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:

```c
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

```c
// 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.

**Brackets**

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

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

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

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

The special characters used as expression operators are covered in the operator section 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,

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

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 $\frac{1}{2}$ decimal digits to mean about 20,000 alternatives and the expression $\frac{3}{4}$ decimal digits to mean more than 20,000 alternatives but less than 100,000 alternatives. The $\frac{1}{2}$ 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.

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

Table 2.7 – Relationships between various representations of precision

<table>
<thead>
<tr>
<th>Decimal Digits</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1,000</td>
</tr>
<tr>
<td>3½</td>
<td>2,000</td>
</tr>
<tr>
<td>3¼</td>
<td>4,000</td>
</tr>
<tr>
<td>4</td>
<td>10,000</td>
</tr>
<tr>
<td>4½</td>
<td>20,000</td>
</tr>
<tr>
<td>4¼</td>
<td>40,000</td>
</tr>
<tr>
<td>5</td>
<td>100,000</td>
</tr>
<tr>
<td>$n$</td>
<td>$10^n$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Pronunciation</th>
<th>Computer Engineering Value</th>
<th>Scientific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>“kay”</td>
<td>$2^{10} = 1024$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>M</td>
<td>“meg”</td>
<td>$2^{20} = 1,048,576$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>G</td>
<td>“gig”</td>
<td>$2^{30} = 1,073,741,824$</td>
<td>$10^9$</td>
</tr>
<tr>
<td>T</td>
<td>“ter”</td>
<td>$2^{40} = 1,099,511,627,776$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>P</td>
<td>“peta”</td>
<td>$2^{50} = 1,125,899,906,843,624$</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>E</td>
<td>“exa”</td>
<td>$2^{60} = 1,152,921,504,606,846,976$</td>
<td>$10^{18}$</td>
</tr>
</tbody>
</table>

Table 2.9 – Common abbreviations for large numbers

8-bit unsigned numbers

A byte contains 8 bits

A byte is 8 bits

where each bit $b_7$, ..., $b_0$ is binary and has the value 1 or 0. We specify $b_7$ as the most significant bit or MSB, and $b_0$ as the least significant bit or LSB. If a byte is used to represent an unsigned number, then the value of the number is

$$N = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0$$

The value of an unsigned byte

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.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Calculation</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01000001</td>
<td>0x41</td>
<td>64 + 1</td>
<td>65</td>
</tr>
<tr>
<td>00010110</td>
<td>0x16</td>
<td>16 + 4 + 2</td>
<td>22</td>
</tr>
<tr>
<td>10000111</td>
<td>0x87</td>
<td>128 + 4 + 2 + 1</td>
<td>135</td>
</tr>
<tr>
<td>11111111</td>
<td>0xff</td>
<td>128 + 64 + 32 + 16 + 8 + 4 + 2 + 1</td>
<td>255</td>
</tr>
</tbody>
</table>

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

\[
\{ 128, 64, 32, 16, 8, 4, 2, 1 \}
\]

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 01100100₂ (which means 64 + 32 + 4).
This operation can be visualized using the table below.

<table>
<thead>
<tr>
<th>Number</th>
<th>Basis</th>
<th>Need it?</th>
<th>Bit</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>128</td>
<td>no</td>
<td>bit7 = 0</td>
<td>none</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>yes</td>
<td>bit6 = 1</td>
<td>100 − 64 = 36</td>
</tr>
<tr>
<td>36</td>
<td>32</td>
<td>yes</td>
<td>bit5 = 1</td>
<td>36 − 32 = 4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>no</td>
<td>bit4 = 0</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>no</td>
<td>bit3 = 0</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>bit2 = 1</td>
<td>4 − 4 = 0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>no</td>
<td>bit1 = 0</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>no</td>
<td>bit0 = 0</td>
<td>none</td>
</tr>
</tbody>
</table>

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 \( 2^n \). \[(2.5)\]

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

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

Defining an unsigned byte in C
8-bit signed numbers

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

\[ N = -128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \]

There are also 256 different signed 8-bit numbers. The smallest signed 8-bit number is -128 and the largest is 127. For example, 10000010₂ is \(-128 + 2\) or \(-126\). Other examples are shown in the following table.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Calculation</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01000001</td>
<td>0x41</td>
<td>64 + 1</td>
<td>65</td>
</tr>
<tr>
<td>00010110</td>
<td>0x16</td>
<td>16 + 4 + 2</td>
<td>22</td>
</tr>
<tr>
<td>10000111</td>
<td>0x87</td>
<td>-128 + 4 + 2 + 1</td>
<td>-121</td>
</tr>
<tr>
<td>11111111</td>
<td>0xff</td>
<td>-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 2.12 – Example conversions of signed 8-bit binary numbers

For the signed 8-bit number system the basis is

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

Notice that the same binary pattern of 11111111₂ 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 \texttt{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.

<table>
<thead>
<tr>
<th>Signed different from unsigned</th>
<th>Signed same as unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/</code> division</td>
<td><code>+</code> addition</td>
</tr>
<tr>
<td><code>*</code> multiplication</td>
<td><code>-</code> subtraction</td>
</tr>
<tr>
<td><code>&gt;</code> greater than</td>
<td><code>==</code> is equal to</td>
</tr>
<tr>
<td><code>&lt;</code> less than</td>
<td>`</td>
</tr>
<tr>
<td><code>&gt;=</code> Greater than or equal to</td>
<td><code>&amp;</code> logical AND</td>
</tr>
<tr>
<td><code>&lt;=</code> Less than or equal to</td>
<td><code>^</code> logical XOR</td>
</tr>
<tr>
<td><code>&gt;&gt;</code> right shift</td>
<td><code>&lt;&lt;</code> left shift</td>
</tr>
</tbody>
</table>

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 10011100₂ (which means -128+16+8+4).
This operation can be visualized using the table below.

<table>
<thead>
<tr>
<th>Number</th>
<th>Basis</th>
<th>Need it?</th>
<th>Bit</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-128</td>
<td>yes</td>
<td>bit7 = 1</td>
<td>-100 – (-128) = 28</td>
</tr>
<tr>
<td>28</td>
<td>64</td>
<td>no</td>
<td>bit6 = 0</td>
<td>none</td>
</tr>
<tr>
<td>28</td>
<td>32</td>
<td>no</td>
<td>bit5 = 0</td>
<td>none</td>
</tr>
<tr>
<td>28</td>
<td>16</td>
<td>yes</td>
<td>bit4 = 1</td>
<td>28 – 16 = 12</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>yes</td>
<td>bit3 = 1</td>
<td>12 – 8 = 4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>bit2 = 1</td>
<td>4 – 4 = 0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>no</td>
<td>bit1 = 0</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>no</td>
<td>bit0 = 0</td>
<td>none</td>
</tr>
</tbody>
</table>

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.

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

A third way to convert negative numbers into binary is to first 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 10011100₂. 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.
To improve the clarity of software, always specify the format of data (signed versus unsigned) when defining or accessing the data.

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

```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

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

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

\[
N = 32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} + 2048 \cdot b_{11} + 1024 \cdot b_{10} + 512 \cdot b_{9} + 256 \cdot b_{8} + 128 \cdot b_{7} + 64 \cdot b_{6} + 32 \cdot b_{5} + 16 \cdot b_{4} + 8 \cdot b_{3} + 4 \cdot b_{2} + 2 \cdot b_{1} + b_{0}
\]

The value of an unsigned word

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 0100₂ or 0x2184 is 8192 + 256 + 128 + 4 or 8580.
Other examples are shown in the following table.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Calculation</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0100 0000 0001</td>
<td>0x0401</td>
<td>1024 + 1</td>
<td>1025</td>
</tr>
<tr>
<td>0000 1100 1010 0000</td>
<td>0x0ca0</td>
<td>2048 + 1024 + 128 + 32</td>
<td>3232</td>
</tr>
<tr>
<td>1000 1110 0000 0010</td>
<td>0x8e02</td>
<td>32768 + 2048 + 1024 + 512 + 2</td>
<td>36354</td>
</tr>
<tr>
<td>1111 1111 1111 1111</td>
<td>0xffff</td>
<td>32768 + 16384 + 8192 + 4096 + 2048 + 1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1</td>
<td>65535</td>
</tr>
</tbody>
</table>

Table 2.15 – Example conversions of unsigned 16-bit binary numbers

For the unsigned 16-bit number system the basis is

\{ 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

```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

\[
N = -32768 \cdot b_{15} + 16384 \cdot b_{14} + 8192 \cdot b_{13} + 4096 \cdot b_{12} + 2048 \cdot b_{11} + 1024 \cdot b_{10} + 512 \cdot b_{9} + 256 \cdot b_{8} + 128 \cdot b_{7} + 64 \cdot b_{6} + 32 \cdot b_{5} + 16 \cdot b_{4} + 8 \cdot b_{3} + 4 \cdot b_{2} + 2 \cdot b_{1} + b_{0}
\]

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 0100₂ or 0xd004 is -32768 + 16384 + 4096 + 4
or -12284. Other examples are shown in the following table.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Calculation</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0100 0000 0001</td>
<td>0x0401</td>
<td>1024 + 1</td>
<td>1025</td>
</tr>
<tr>
<td>0000 1100 1010 0000</td>
<td>0x0ca0</td>
<td>2048 + 1024 + 128 + 32</td>
<td>3232</td>
</tr>
<tr>
<td>1000 1110 0000 0010</td>
<td>0x8e02</td>
<td>-32768 + 2048 + 1024 + 512 + 2</td>
<td>-31742</td>
</tr>
<tr>
<td>1111 1111 1111 1111</td>
<td>0xffff</td>
<td>-32768+16384+8192+4096+2048+1024+512+256+128+64+32+16+8+4+2+1</td>
<td>-1</td>
</tr>
</tbody>
</table>

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

```c
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:

```c
// 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;
```

typedefs for signed and unsigned data
Big- and Little-Endian

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the big-endian approach that stores the most significant part first. 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 $0x03e8$ at locations $0x50, 0x51$, then

```
Big-endian          Little-endian
address  contents   address  contents
0x0050  0x030051  0xe8
```

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

```
Big-endian          Little-endian
address  contents   address  contents
0x0050  0x120051  0x340052  0x560053  0x78
```

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 nondivisible 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.

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0050</td>
<td>0x36</td>
</tr>
<tr>
<td>0x0051</td>
<td>0x38</td>
</tr>
<tr>
<td>0x0052</td>
<td>0x31</td>
</tr>
<tr>
<td>0x0053</td>
<td>0x32</td>
</tr>
</tbody>
</table>

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 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:

- **False** to be all zeros, and
- **True** to be any nonzero value.

Many programmers add the following macros to their code

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

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

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

Decimal Numbers

Decimal numbers are written as a sequence of decimal digits (0 through 9). The number may be preceded by a plus or minus sign or followed by 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0 to 255</td>
<td>8 bits</td>
<td>0, 10, 123</td>
</tr>
<tr>
<td>char</td>
<td>-128 to 127</td>
<td>8 bits</td>
<td>-123, 0, 10, +10</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to 65536U</td>
<td>16 bits</td>
<td>0, 2000, 2000U, 50000U</td>
</tr>
<tr>
<td>int</td>
<td>-32768 to 32767</td>
<td>16 bits</td>
<td>-1000, 0, 1000, +20000</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0 to 65536U</td>
<td>16 bits</td>
<td>0, 2000, 2000U, 50000U</td>
</tr>
<tr>
<td>short</td>
<td>-32768 to 32767</td>
<td>16 bits</td>
<td>-1000, 0, 1000, +20000</td>
</tr>
<tr>
<td>long</td>
<td>-2147483648L to 2147483647L</td>
<td>32 bits</td>
<td>-1234567L, 0L, 1234567L</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Freescale MC9S12</th>
<th>Intel x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>char</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>int</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>long</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

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.
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

```c
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

```c
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0 to 0377</td>
<td>8 bits</td>
<td>0, 010, 0123</td>
</tr>
<tr>
<td>char</td>
<td>-0200 to 0177</td>
<td>8 bits</td>
<td>-0123, 0, 010, +010</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to 0177777U</td>
<td>16 bits</td>
<td>0, 02000, 0150000U</td>
</tr>
<tr>
<td>int</td>
<td>-077777 to 77777</td>
<td>16 bits</td>
<td>-01000, 0, 01000, +020000</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0 to 0177777U</td>
<td>16 bits</td>
<td>0, 02000, 0150000U</td>
</tr>
<tr>
<td>short</td>
<td>-077777 to 77777</td>
<td>16 bits</td>
<td>-01000, 0, 01000, +020000</td>
</tr>
<tr>
<td>long</td>
<td>-017777777777L to 017777777777L</td>
<td>32 bits</td>
<td>-01234567L, 0L, 01234567L</td>
</tr>
</tbody>
</table>

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.
Hexadecimal Numbers

The hexadecimal number system uses base 16 as opposed to our regular decimal number system that uses base 10. Like the octal format, the hexadecimal format is also a convenient mechanism for 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.

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

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.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Binary format</th>
<th>Hexadecimal format</th>
<th>Decimal format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freescale assembly language</td>
<td>%01111010</td>
<td>$7a</td>
<td>122</td>
</tr>
<tr>
<td>Intel and TI assembly language</td>
<td>0111010b</td>
<td>7ah</td>
<td>122</td>
</tr>
<tr>
<td>C language</td>
<td>-</td>
<td>0x7a</td>
<td>122</td>
</tr>
</tbody>
</table>

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.

![Binary to Hexadecimal Conversion](image)

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.

![Hexadecimal to Binary Conversion](image)

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.
The range of a hexadecimal number depends on the data type as shown in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0x00 to 0xff</td>
<td>8 bits</td>
<td>0x01, 0x3a, 0xb3</td>
</tr>
<tr>
<td>char</td>
<td>-0x80 to 0x7f</td>
<td>8 bits</td>
<td>-0x01, 0x3a, -0x7b</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0x0000 to 0xffff</td>
<td>16 bits</td>
<td>0x22, 0Xabcd, 0xf0a6</td>
</tr>
<tr>
<td>int</td>
<td>-0x8000 to 0x7fff</td>
<td>16 bits</td>
<td>-0x22, 0X0, +0x70A6</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0x0000 to 0xffff</td>
<td>16 bits</td>
<td>0x22, 0Xabcd, 0xf0a6</td>
</tr>
<tr>
<td>short</td>
<td>-0x8000 to 0x7fff</td>
<td>16 bits</td>
<td>-0x1234, 0x0, 0x7abc</td>
</tr>
<tr>
<td>long</td>
<td>-0x80000000 to 0x7fffffff</td>
<td>32 bits</td>
<td>-0x1234567, 0xABCDEF</td>
</tr>
</tbody>
</table>

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

```c
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

```assembly
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.

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:

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

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.

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

```c
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.

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:

```c
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

```assembly
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:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>newline, linefeed</td>
<td>0x0a = 10</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
<td>0x09 = 9</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
<td>0x08 = 8</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
<td>0x0c = 12</td>
</tr>
<tr>
<td>\a</td>
<td>bell</td>
<td>0x07 = 7</td>
</tr>
<tr>
<td>\r</td>
<td>return</td>
<td>0x0d = 13</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
<td>0x0b = 11</td>
</tr>
<tr>
<td>\0</td>
<td>null</td>
<td>0x00 = 0</td>
</tr>
<tr>
<td>&quot;</td>
<td>ASCII double quote</td>
<td>0x22 = 34</td>
</tr>
<tr>
<td>\</td>
<td>ASCII back slash</td>
<td>0x5c = 92</td>
</tr>
<tr>
<td>'</td>
<td>ASCII single quote</td>
<td>0x27 = 39</td>
</tr>
</tbody>
</table>

**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:

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

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

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

### 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.

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

```c
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 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.

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.
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.

```
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 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

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.

```c
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

```assembly
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

```assembly
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

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.

```c
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:

```c
    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.

```c
    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 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

```c
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.

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 (deallocating).

```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.

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

Figure 2.4 – MC9S12 implementation of three local variables – step 2
### Figure 2.5 – MC9S12 implementation of three local variables – step 3

```
address          data
0x0BE0          D  1000
0x0BE2          IX
0x0BE4          IY
0x0BE8          SP  0x0BE4

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

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

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

RegPC ───> 0x0BEA
```

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

```
address          data
0x0BE0          D  1000
0x0BE2          IX
0x0BE4          IY
0x0BE8          SP  0x0BE4

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

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

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

RegPC ───> 0x0BEA
```
Figure 2.7 – MC9S12 implementation of three local variables – step 5

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
    std 2,sp
    addd #12000 ; 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

Figure 2.10 – MC9S12 implementation of three local variables – step 8
Figure 2.11 – MC9S12 implementation of three local variables – step 9

Address: 0x0BE8
Data: return address

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

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

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

Address: 0x0BE8
Data: return address

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

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

```c
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)

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

.bss
TheGlobal
```
Externals

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

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

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

```c
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

```assembly
.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.

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

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

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

```c
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**
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.

```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.

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.
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.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Comment</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned char uc;</code></td>
<td>8-bit unsigned number</td>
<td>0 to +255</td>
</tr>
<tr>
<td><code>char c1, c2, c3;</code></td>
<td>three 8-bit signed numbers</td>
<td>-128 to +127</td>
</tr>
<tr>
<td><code>unsigned int ui;</code></td>
<td>16-bit unsigned number</td>
<td>0 to +65535</td>
</tr>
<tr>
<td><code>int i1, i2;</code></td>
<td>two 16-bit signed numbers</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td><code>unsigned short us;</code></td>
<td>16-bit unsigned number</td>
<td>0 to +65535</td>
</tr>
<tr>
<td><code>short s1, s2;</code></td>
<td>two 16-bit signed numbers</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td><code>long l1, l2, l3, l4;</code></td>
<td>four signed 32-bit integers</td>
<td>-2147483648L to 2147483647L</td>
</tr>
<tr>
<td><code>float f1, f2;</code></td>
<td>two 32-bit floating-point numbers</td>
<td>not recommended</td>
</tr>
<tr>
<td><code>double d1, d2;</code></td>
<td>two 64-bit floating-point numbers</td>
<td>not recommended</td>
</tr>
</tbody>
</table>

Table 2.24 – Variable declarations

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.

<table>
<thead>
<tr>
<th>Storage Class</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>auto</code></td>
<td>automatic, allocated on the stack</td>
</tr>
<tr>
<td><code>extern</code></td>
<td>defined in some other program file</td>
</tr>
<tr>
<td><code>static</code></td>
<td>permanently allocated</td>
</tr>
<tr>
<td><code>register</code></td>
<td>attempt to implement an automatic using a register instead of on the stack</td>
</tr>
</tbody>
</table>

Table 2.25 – Variable storage classes
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile</td>
<td>can change value by means other than the current program</td>
</tr>
<tr>
<td>const</td>
<td>fixed value, defined in the source code and can not be changed during execution</td>
</tr>
<tr>
<td>unsigned</td>
<td>range starts with 0, includes only positive values</td>
</tr>
<tr>
<td>signed</td>
<td>range includes both negative and positive values</td>
</tr>
</tbody>
</table>

Table 2.26 – Variable modifiers

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.

```cpp
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.

```cpp
void NotLegalFunction(const short Count)
{
    while (Count)
    {
        SCI_OutChar(13);
        Count--;  // this operation is illegal
        Count--;
    }
}
```
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.

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

**Character Variables**

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.

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.

```c
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:

- 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.
Initialization of variables and constants

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

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

```c
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

```c
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,

```c
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

```c
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,

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

```c
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

```c
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

```c
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

```c
char *NamePt = "John";
```

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.

```c
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

```c
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
int I;
void main(void)
{
    I = 95;
}

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

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:

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

Table 2.27 – Attributes of variables stored in memory

Summary of Variable Lifetimes

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

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

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.

Precedence and Associativity

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

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

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

\[ y = 254 \times x / 100; \]

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:

\[ y = 254 \times (x / 100); \]
The proper approach is to multiply first then divide. To multiply first we must guarantee that the product $254 \times x$ will not overflow the precision of the computer. How do we know what precision the compiler used for the intermediate result $254 \times x$? 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:

$$y = \frac{254 \times x}{100};$$

The issues of precedence and associativity were explained in an earlier section. Precedence defines the evaluation order. For example the expression $3 + 4 \times 2$ 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.
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:

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>binary complement</td>
<td>~0x1234</td>
<td>0xEDCB</td>
</tr>
<tr>
<td>!</td>
<td>logical complement</td>
<td>!flag</td>
<td>flip 0 to 1 and not zero to 0</td>
</tr>
<tr>
<td>&amp;</td>
<td>address of</td>
<td>&amp;data</td>
<td>address in memory where data is stored</td>
</tr>
<tr>
<td>-</td>
<td>negate</td>
<td>-100</td>
<td>negative 100</td>
</tr>
<tr>
<td>+</td>
<td>positive</td>
<td>+100</td>
<td>100</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++data</td>
<td>data=data+1, then result is data</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--data</td>
<td>data=data-1, then result is data</td>
</tr>
<tr>
<td>*</td>
<td>reference</td>
<td>*pt</td>
<td>16-bit information pointed to by pt</td>
</tr>
</tbody>
</table>

Table 2.29 – Unary prefix operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Postincrement</td>
<td>data++</td>
<td>result is data, then data=data+1</td>
</tr>
<tr>
<td>--</td>
<td>Postdecrement</td>
<td>data--</td>
<td>result is data, then data=data-1</td>
</tr>
</tbody>
</table>

Table 2.30 – Unary postfix operators
Binary operators

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>100 + 300</td>
<td>400</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>100 - 300</td>
<td>~200</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>10 * 300</td>
<td>3000</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>123 / 10</td>
<td>12</td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
<td>123 % 10</td>
<td>3</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>102 &lt;&lt; 2</td>
<td>408</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>102 &gt;&gt; 2</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 2.31 – Binary arithmetic operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>0x1234 &amp; 0x00FF</td>
<td>0x0034</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>0x1234</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>0x1234 ^ 0x00FF</td>
<td>0x12CB</td>
</tr>
</tbody>
</table>

Table 2.32 – Binary bitwise logical operators
The binary Boolean operators take two Boolean inputs and give a single Boolean result.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>0 &amp;&amp; 1</td>
<td>0 (false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
</tbody>
</table>

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).

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>is equal to</td>
<td>100 == 200</td>
<td>0 (false)</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
<td>100 != 200</td>
<td>1 (true)</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>100 &lt; 200</td>
<td>1 (true)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>100 &lt;= 200</td>
<td>1 (true)</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>100 &gt; 200</td>
<td>0 (false)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>100 &gt;= 200</td>
<td>0 (false)</td>
</tr>
</tbody>
</table>

Table 2.34 – Binary relational operators

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

```c
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.

```c
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.

```c
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
}
```

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.

```c
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
2.100

Expression Types and Explicit Casting

We saw earlier that numbers are represented in the computer using a wide range of formats. A list of these formats is given in Table 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
<th>Example Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0 to 255</td>
<td>8-bits</td>
<td>unsigned char uc;</td>
</tr>
<tr>
<td>char</td>
<td>-127 to 127</td>
<td>8-bits</td>
<td>char sc;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to 65535U</td>
<td>16-bits</td>
<td>unsigned int ui;</td>
</tr>
<tr>
<td>int</td>
<td>-32768 to 32767</td>
<td>16-bits</td>
<td>int si;</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0 to 65535U</td>
<td>16-bits</td>
<td>unsigned short us;</td>
</tr>
<tr>
<td>short</td>
<td>-32768 to 32767</td>
<td>16-bits</td>
<td>short ss;</td>
</tr>
<tr>
<td>long</td>
<td>-2147483648L to 2147483647L</td>
<td>32-bits</td>
<td>long sl;</td>
</tr>
</tbody>
</table>

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:

- 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.

<table>
<thead>
<tr>
<th>Type1</th>
<th>Type2</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>unsigned short</td>
<td>uc + us is of type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned short</td>
</tr>
<tr>
<td>unsigned char</td>
<td>short</td>
<td>uc + ss is of type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short</td>
</tr>
<tr>
<td>unsigned char</td>
<td>long</td>
<td>uc + sl is of type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>char</td>
<td>short</td>
<td>sc + ss is of type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short</td>
</tr>
<tr>
<td>char</td>
<td>long</td>
<td>sc + sl is of type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>unsigned short</td>
<td>long</td>
<td>us + sl is of type</td>
</tr>
<tr>
<td>short</td>
<td>long</td>
<td>ss + sl is of type</td>
</tr>
</tbody>
</table>

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
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.

```c
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

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.

```c
// 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

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.

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

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

```
Expr1 ? Expr2 : Expr3
```

The first input parameter is an expression, Expr1, which yields a Boolean (0 for false, not zero for true). Expr2 and Expr3 return values that are regular numbers. The selection operator will return the result of Expr2 if the value of Expr1 is true, and will return the result of Expr3 if the value of Expr1 is false. The type of the expression is determined by the types of Expr2 and Expr3. If Expr2 and Expr3 have different types, then the usual promotion is applied. The resulting 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.

```c
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
is important to remember that arithmetic operations (addition, subtraction, multiplication, division, and shifting) have constraints when performed with finite precision on a microcomputer. An overflow error occurs when the result of an arithmetic operation can not fit into the finite precision of the result. We will study addition and subtraction operations in detail, but the techniques for dealing with overflow and underflow will apply to the other arithmetic operations as well. We will consider two approaches:

- avoiding the error
- detecting the error then correcting the result

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

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

<table>
<thead>
<tr>
<th>bit</th>
<th>name</th>
<th>Meaning after addition or subtraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>negative</td>
<td>result is negative</td>
</tr>
<tr>
<td>Z</td>
<td>zero</td>
<td>result is zero</td>
</tr>
<tr>
<td>V</td>
<td>overflow</td>
<td>signed overflow</td>
</tr>
<tr>
<td>C</td>
<td>carry</td>
<td>unsigned overflow</td>
</tr>
</tbody>
</table>

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.

![Figure 2.13 – 8-bit unsigned addition](image)

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).
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.**

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.
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).

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.  

\[ (2.13) \]
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 if 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.

\[
\begin{array}{|c|c|}
\hline
N & R_7 \\
\hline
Z & R_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0 \\
\hline
V & X_7 \cdot M_7 \cdot R_7 \\
& + \overline{X_7} \cdot M_7 \cdot R_7 \\
\hline
C & X_7 \cdot M_7 \\
& + M_7 \cdot \overline{R_7} \\
& + X_7 \cdot \overline{R_7} \\
\hline
\end{array}
\]

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.

\[
\begin{array}{|c|c|}
\hline
N & R_7 \\
\hline
Z & R_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0 \\
\hline
V & X_7 \cdot M_7 \cdot \overline{R_7} \\
& + \overline{X_7} \cdot M_7 \cdot R_7 \\
\hline
C & \overline{X_7} \cdot M_7 \\
& + M_7 \cdot \overline{R_7} \\
& + \overline{X_7} \cdot \overline{R_7} \\
\hline
\end{array}
\]

Table 2.39 – Condition code bits after an 8-bit subtraction operation
Ignoring overflow (signed or unsigned) can result in significant errors. \hspace{2cm} (2.14)

Computers have two sets of conditional branch instructions (if statements) which make program decisions based on either the C or V bit. \hspace{2cm} (2.15)

An error will occur if you use unsigned conditional branch instructions (if statements) after operating on signed numbers, and vice-versa. \hspace{2cm} (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.
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.

![Flowchart](image)

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

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

<table>
<thead>
<tr>
<th>decimal</th>
<th>8-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>-96</td>
<td>1010 0000</td>
<td>1111 1111 1010 0000</td>
</tr>
<tr>
<td>-24</td>
<td>-0100 0000</td>
<td>-0000 0000 0100 0000</td>
</tr>
<tr>
<td>-160</td>
<td>0110 0000</td>
<td>1111 1111 0110 0000</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>signed add</th>
<th>signed sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promote $X$ to $X_{16}$</td>
<td>Promote $X$ to $X_{16}$</td>
</tr>
<tr>
<td>Promote $M$ to $M_{16}$</td>
<td>Promote $M$ to $M_{16}$</td>
</tr>
<tr>
<td>$R_{16} = X_{16} + M_{16}$</td>
<td>$R_{16} = X_{16} + M_{16}$</td>
</tr>
<tr>
<td>underflow $R_{16} &lt; -128$</td>
<td>underflow $R_{16} &lt; -128$</td>
</tr>
<tr>
<td>$R = -128$</td>
<td>$R = -128$</td>
</tr>
<tr>
<td>$R = R_{16}$</td>
<td>$R = R_{16}$</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

**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.)

Embedded Software 2014
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.

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.
2.114

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

```c
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

```c
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

\[
\{ \text{ObjectDeclaration?... Statement?... } \}
\]

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.

```c
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 \texttt{goto} and \texttt{switch} statements (below) could violate this rule.
The if Statement

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

```plaintext
if ( ExpressionList )
    Statement1
```
or

```plaintext
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.

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

Listing 2.55 – Example if statement
A 3-wide median filter can be designed using **if-else** conditional statements.

```c
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.

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

![Diagram showing conditional logic](image-url)
The switch Statement

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

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

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

The case prefix has the form

```
case ConstantExpression:
```

and the default prefix has the form

```
default:
```

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

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

```c
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.

```c
#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.

```c
// 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
```

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

```c
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

```c
#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**
2.122

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

```
continue;
```

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

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

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

```plaintext
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,

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

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

```plaintext
for (i = 4; i >= 0; --i)
    array[i] = 0;
```
2.124

or a little more efficiently as

```c
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

```c
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`. 

Embedded Software 2014
The **do** Statement

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

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

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

1. **Statement** is executed.
2. 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

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

or as

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

or as

```c
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

```c
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.

Take, for example, the statement

```c
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

```
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.

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.
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

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

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

Other examples include an array or a structure name. As we shall see in 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.

```c
// 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."
2.132

**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.

---

**Figure 2.21 – MC9S12 example of pointer references – step 1**

---

Example of legitimate pointer references in C

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0810</td>
<td>pt</td>
</tr>
<tr>
<td>0x0812</td>
<td>data</td>
</tr>
<tr>
<td>0x0814</td>
<td>buffer[0]</td>
</tr>
<tr>
<td>0x0816</td>
<td>buffer[1]</td>
</tr>
<tr>
<td>0x0818</td>
<td>buffer[2]</td>
</tr>
<tr>
<td>0x081A</td>
<td>buffer[3]</td>
</tr>
<tr>
<td>0x081C</td>
<td></td>
</tr>
</tbody>
</table>

Example of legitimate pointer references in C

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

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

.bss
pt
data
buffer
```
Figure 2.22 – MC9S12 example of pointer references – step 2

Figure 2.23 – MC9S12 example of pointer references – step 3
Address data
0x0810 pt D 0x1234
0x0812 data IX
0x0814 buffer[0] IY 0x0816
0x0816 buffer[1] SP
0x0818 buffer[2]
0x081A buffer[3]
0x081C

Figure 2.24 – MC9S12 example of pointer references – step 4

Address data
0x0810 pt D 0x1234
0x0812 data IX
0x0814 buffer[0] IY 0x0816
0x0816 buffer[1] SP
0x0818 buffer[2]
0x081A buffer[3]
0x081C

Figure 2.25 – MC9S12 example of pointer references – step 5
Figure 2.26 – MC9S12 example of pointer references – step 6

Figure 2.27 – MC9S12 example of pointer references – step 7
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.

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

![Figure 2.29 – The MC9S12 has 16-bit address registers, IX, IY, SP and PC](image)

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 erase (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.

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

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.

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.

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 \( \text{ptr} \) is a signed 16-bit integer pointer, then \( \text{ptr} + 1 \) also points to a signed 16-bit integer.

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 \( *\text{ptr} + 1 \) is the same as \( (*\text{ptr}) + 1 \) and not \( *\text{(ptr + 1)} \). 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

```c
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.

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.
2.143

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

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

Embedded Software 2014
A FIFO_Put() followed by a FIFO_Get()...

Two FIFO_Put() operations…

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

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

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

A FIFO_Put() followed by a FIFO_Get()
Figure 2.35 – FIFO example showing the wrapping of pointers – step 6

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

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

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.
```c
#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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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.

```c
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):

```c
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:

```c
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.
# 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 \texttt{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 \texttt{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.

\begin{verbatim}
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 *);
\end{verbatim}

\textbf{Listing 2.72 – Prototypes for string functions}

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

\begin{verbatim}
void *memchr(void *p, int c, size_t n);
\end{verbatim}

Starting in memory at address \( p \), \texttt{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.

\begin{verbatim}
int memcmp(void *p, void *q, size_t n);
\end{verbatim}

Assuming the two pointers are directed at 8-bit data blocks of size \( n \), \texttt{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`.

```c
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.

```c
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:

```c
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`).

```c
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.

```c
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:

\[
n = \text{strncmp}("MC9S12A256", "MC9S12A512", 7);
\]

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 pointed 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:

\[
\text{pt} = \text{strpbrk}("label: ldaa 10,x ;comment", " ;:*/n\t\l")
\]

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.

```c
// 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:

```c
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:

```c
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:

```c
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.

```c
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:

```c
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:

```c
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:

```c
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
embedded software 2014

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:

```c
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.

```c
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.

```c
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:

```c
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 \texttt{fsm[3]} structure shown previously. The state machine is illustrated below, along with the program.

![Finite state machine diagram]

\textbf{Figure 2.38 – Finite state machine}

\begin{verbatim}
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;
            }
    }
}
\end{verbatim}

\textbf{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.

```c
#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.

```c
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.

```c
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](image-url)
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.

```c
#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.

```c
#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.

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 letters 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.

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.

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.
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.

```c
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 twentieth= {'P',0,&twentyfirst};
TNode nineteenth= {'Y',0,&twentieth};
TNode eighteenth= {'G',0,&nineteenth};
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= {'I',0,&eighth};
TNode sixth= {'N',0,&seventh};
TNode fifth= {'T',0,&sixth};
TNode fourth= {'O',0,&fifth};
TNode third= {'A',0,&fourth};
TNode second= {'T',0,&third};
TNode root= {'E',0,&second};
```
/ **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
                    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;
    FNode 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.
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:

```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:

```c
// 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:

```c
((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.
Normally we place function declarations in the header file. We should add
comments that explain what the function does.

To illustrate some options when declaring functions, alternative declarations of
these same five functions are given below:

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

```c
// 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 \textbf{int} input that returns an \textbf{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.

```c
#include "LCD.h"
#include "COP.h"
#include "Timer.h"

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

    COP_Init();
    LCD_Init();
}
```

Embedded Software 2014
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:

```c
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:

\[
\text{Name}(\text{parameter list})
\]

where \text{Name} is the name of the function to be called. The \text{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:

\[
\text{add3}(-\text{counter}, \text{time} + 5, 3);
\]

is legal. It calls \text{add3()}, passing it three arguments: \text{--counter}, \text{time} + 5, and 3. Since this call is not part of a larger expression, the value that \text{add3()} returns will be ignored. As a better example, consider:

\[
y = \text{add3}(-\text{counter}, \text{time} + 5, 3);
\]
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.

```c
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
}

cchar 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`).

```c
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:

```c
func(y = x + 1, 2 * y);
```

where the first argument has the value \(x + 1\) and the second argument has the value \(2 \times (x + 1)\).

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.

```c
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.

```c
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` constant is replaced with a `void (*CmdPt)(void);` function pointer. The three general functions `DoStop()`, `DoTurn()`, and `DoBend()` are also added.

```c
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.

```c
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.

```c
// 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, TFU(pullup) timer/4 (500ns)
    OutString("Enter a single letter followed by <enter>");
    while (1)
    {
        OutString(">");
        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:

```c
#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
#else
... // ABC and DEF
#endif
#endif
#else
... // not ABC
#ifdef HIJ
... // not ABC but HIJ
#endif
#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:

```c
#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:

```c
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.

```c
asm cli
```

A better way is to `#define` macros:

```c
#define INTR_ON() asm cli
#define INTR_OFF() asm sei
```
The following function runs with interrupts disabled.

```c
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.

```c
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**
Lecture 5 – Interrupts

Interrupts, Interrupt service routines, Hardware interrupts, Interrupt vectors and priority, Exceptions, Threads, Foreground and background threads, Re-entrant programming.

Introduction

You are studying at your desk at home. The phone rings (an interrupt). You stop studying and answer the phone (you accept the interrupt). It is your friend, who wants to know the URL for a particular Freescale datasheet relating to the MC9S12 so she can look up some information required to complete a laboratory assignment. You give her the URL (you process the interrupt request immediately). You then hang up and go back to studying. Note that the additional time it will take you to complete your study is miniscule, yet the amount of time for your friend to complete her task may be significantly reduced. This simple example clearly illustrates how interrupts can drastically improve response time in a real-time system.

Interrupts are an essential feature of a microcontroller. They enable the software to respond, in a timely fashion, to internal and external hardware events. For example, the reception and transmission of bytes via the SCI is more efficient (in terms of processor time) using interrupts, rather than using a polling method. Performance is improved because tasks can be given to hardware modules which “report back” when they are finished.

Using interrupts requires that we first understand how a CPU processes an interrupt so that we can configure our software to take advantage of them.
5.2

Interrupts

An interrupt is an event triggered inside the microcontroller, either by internal or external hardware, that initiates the automatic transfer of software execution to an interrupt service routine (ISR). On completion of the ISR, software execution returns to the next instruction that would have occurred without the interrupt.

![Diagram of interrupt process]

Figure 5.1

A thread is defined as a sequence of instructions that has its own program counter, stack and registers; it shares its address space and system resources with other threads. By contrast, a process has its own virtual address space (stack, data, code) and system resources (e.g. open files). Processes are normally used in systems with an operating system, whereas threads are easily implemented in simple embedded systems using interrupt service routines.

In the MC9S12 microcontroller, the hardware automatically pushes the contents of the internal registers onto the stack, thus creating the correct environment for a new thread invoked by an ISR.
Using Interrupts

Each potential interrupt source has a separate arm bit, e.g. RIE (the SCI receive interrupt enable bit). The software must set the arm bits for those devices from which it wishes to accept interrupts, and deactivate the arm bits within those devices from which interrupts are not to be allowed. After reset, all the interrupt arm bits are set to deactivate the corresponding interrupt.

Each potential interrupt source has a separate flag bit, e.g. RDRF (the SCI receive data register full flag). The hardware sets the flag when it wishes to request an interrupt. The software must clear the flag in the ISR to signify it has handled the interrupt request, and to allow the device to again trigger an interrupt.

The global interrupt enable bit, I, which is in the condition code register (CCR) is used to enable or disable all armed interrupts. Software enables all armed interrupts by setting $I = 0$, (“asm cli” in C), and disables all interrupts by setting $I = 1$ (“asm sei” in C). $I = 1$ does not dismiss the interrupt requests, rather it postpones them.

Three conditions must be true simultaneously for an interrupt to occur:

- The individual arm bit for an interrupt must be set in software.
- When it is convenient, the software will enable all armed interrupts.
- The hardware state changes from “busy” to “ready” and sets a flag.

The following figure shows the hardware arrangement for interrupt generation.

![Figure 5.2](image-url)
5.4

Interrupt Processing

When an interrupt occurs, the following sequence is followed.

1. The execution of the main program is suspended by the hardware:
   - the current instruction is finished
   - all the registers are pushed onto the stack
   - the vector address is retrieved from high memory and placed in the PC
   - the I bit in the CCR is set, disabling further interrupts

2. The interrupt service routine, or foreground thread, is executed. The ISR:
   - performs the necessary operations
   - clears the flag that requested the interrupt
   - communicates using global variables

3. The main program is resumed when the ISR executes the rti instruction:
   - Hardware pulls all the registers from the stack, including the PC, so that the program continues from the point where it was interrupted.

Interrupt Polling

Some interrupts share the same interrupt vector. For example, the reception and transmission of a byte via the SCI leads to just one interrupt, and there is one vector associated with it. In Figure 5.2, the two interrupt sources are ORed together to create one interrupt request. In such cases, the ISR is responsible for polling the status flags to see which event actually triggered the interrupt. Care must be taken because both flags may be set, and both hardware events must be serviced by the software.
Interrupt Service Routines (ISRs)

An interrupt service routine (ISR) is a section of code specifically designed to respond to the interrupt request. When the CPU begins to service an interrupt, the instruction queue is refilled, a return address calculated, and then the return address and the contents of the CPU registers are automatically stacked as shown below:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP – 2</td>
<td>(\text{RTN}_H : \text{RTN}_L)</td>
</tr>
<tr>
<td>SP – 4</td>
<td>(\text{Y}_H : \text{Y}_L)</td>
</tr>
<tr>
<td>SP – 6</td>
<td>(\text{X}_H : \text{X}_L)</td>
</tr>
<tr>
<td>SP – 8</td>
<td>(\text{B} : \text{A})</td>
</tr>
<tr>
<td>SP – 9</td>
<td>(\text{CCR})</td>
</tr>
</tbody>
</table>

Table 5.1 – Stacking Order on Entry to Interrupts

After the \(\text{CCR}\) is stacked, the \(I\) bit (and the \(X\) bit, if an \(\overline{XIRQ}\) interrupt service request caused the interrupt) is set to prevent other interrupts from disrupting the interrupt service routine. Execution continues at the address pointed to by the vector for the highest-priority interrupt that was pending at the beginning of the interrupt sequence – this is the interrupt service routine.

The body of an interrupt service routine varies according to the source of the interrupt. For an interrupt service routine written to handle external events, they typically respond to the interrupt by retrieving or sending external data, e.g. the reception of a byte of data via the SCI is normally handled via an ISR which places the received byte into a FIFO for later processing by the main function.

At the end of the interrupt service routine, an \(\text{rti}\) instruction restores context from the stacked registers, and normal program execution resumes (which could also be recognition of another interrupt).
Declaring Interrupt Service Routines in C

In CodeWarrior, an interrupt service routine is declared with the non-ANSI C keyword `interrupt`, as well as a number corresponding to an entry in the vector table. For example, to declare an ISR for SCI0, you would use:

```c
void interrupt 20 SCI0_ISR(void) {
    /* code goes here */
}
```

This syntax tells the compiler that the function is an interrupt function (i.e. it should be terminated with `rti` rather than `rts`) and initializes the corresponding entry in the vector table. For the MC9S12, the reset vector is vector number 0 (at address 0xFFFE), vector number 1 is located just before the vector 0 (at address 0xFFFC), and so on.

Enabling and Disabling Interrupts

Interrupts can be enabled and disabled with the macros defined in the `hidef.h` file which you can include by placing `#include <hidef.h>` in your main file. The macros are:

```c
#define EnableInterrupts   __asm cli;
#define DisableInterrupts  __asm sei;
```

Before the main loop of your program, but after setting up various modules, you will need to enable interrupts, as they are disabled by default.

Interrupt Latency

Interrupts cannot disturb an instruction in progress, and thus are only recognized between the execution of two instructions (apart from special instructions on the MC9S12 which are designed to be interrupted). Therefore the maximum latency from interrupt request to completion of the hardware response consists of the execution time of the slowest instruction plus the time required to complete the memory transfers required by the hardware response.
Example

Suppose we wish to make a simple application using the real-time interrupt to generate pulses on Port T, bits 0 and 1 (the pulses on these particular output pins could be used to keep track of the elapsed time by an external counter, or for viewing interrupt processing time on a DSO, for example).

The code below shows a simple scheme that also shows the timing operation of the main loop.

```c
void RTI_Init(void)
{
    // PortT bits 1 and 0 are outputs
    PTT = 0;
    DDRT_DDRT0 = 1;
    DDRT_DDRT1 = 1;
    // period = 125 ns * 4096 * 4
    RTICTL = 0x33;
    // Enable RTI
    CRGINT_RTIE = 1;
    // Interrupt counter
    Count = 0;
    // Foreground is ready
    Ack = 1;
    // Enable interrupts
    asm cli;
}

void interrupt 7 RTI_ISR(void)
{
    // Acknowledge interrupt, clear RTIF
    CRGFLG_RTIF = 1;
    // Set bit 0
    PTT_PTT0 = 1;
    // Software handshake - means RTI happened
    if (Ack == 1)
    {
        Ack = 0;
        // Number of interrupts
        Count++;
        PTT_PTT0 = 0;
    }
}

void main(void)
{
    RTI_Init();
    for (;;)
    {
        if (Ack == 0)
        {
            Ack = 1;
            PTT ^= PTT_PTT1_MASK;
        }
    }
}
```

Listing 5.1 – Code for a Real-time Clock
Figure 5.3 – Flowchart for Real-Time Interrupt Program
Exceptions

Exceptions are events that require processing outside the normal flow of instruction execution. They generalize the concept of an interrupt – an interrupt is just a particular type of exception.

On the MC9S12, exceptions include resets, an unimplemented opcode trap, a software interrupt instruction, external interrupts and internal interrupts. Each exception has an associated 16-bit vector, which points to the memory location where the routine that handles the exception is located. As shown below, the vectors are stored in the upper 128 bytes of the standard 64-Kbyte address map.

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFE-$FFFF</td>
<td>System Reset</td>
</tr>
<tr>
<td>$FFFC-$FFFFD</td>
<td>Clock Monitor Reset</td>
</tr>
<tr>
<td>$FFFA-$FFFFB</td>
<td>COP Reset</td>
</tr>
<tr>
<td>$FFF8-$FFF9</td>
<td>Unimplemented Opcode Trap</td>
</tr>
<tr>
<td>$FFF6-$FFF7</td>
<td>Software Interrupt Instruction (SWI)</td>
</tr>
<tr>
<td>$FFF4-$FFF5</td>
<td>XIRQ Signal</td>
</tr>
<tr>
<td>$FFF2-$FFF3</td>
<td>IRQ Signal</td>
</tr>
<tr>
<td>$FFC0-$FFF1</td>
<td>Device-Specific Interrupt Sources</td>
</tr>
</tbody>
</table>

Table 5.2 – MC9S12 Exception Vector Map

The six highest vector addresses are used for resets and unmaskable interrupt sources. The remaining vectors are used for maskable interrupts. All vectors must be programmed to point to the address of the appropriate service routine.

The higher the vector address, the higher the priority of the interrupt.

Table 5-1 of the Freescale document *MC9S12DP512 Device Guide* lists the entire Vector Table for the MC9S12A512 device.
5.10

Foreground / Background Threads

In many systems where response time is critical, it is common to organize the program as a foreground / background system, as shown below.

Most of the actual work is performed in the “foreground”, implemented as one or more interrupt service routines with each ISR processing a particular hardware event. This allows the system to respond to external events with a predictable amount of latency. To the extent that the external events are independent, there may be little or no communication between the various ISRs.

The main program performs the necessary initialization and then enters the “background” portion of the program, which is often nothing more than a simple loop that processes non-critical tasks and waits for interrupts to occur. Examples of background processing include: processing data from an input device, creating data for an output device, making calculations based on analog-to-digital conversion results, determining the next digital-to-analog output, and updating a display seen by human eyes.
Serial Communication Interface using Interrupts

Consider the common case of an application that uses the serial communication interface (SCI). The SCI hardware receives characters at an asynchronous rate. In order to avoid loss of data in periods of high activity, the characters need to be stored in a FIFO buffer. The background task (main program) can process the characters at a rate which is independent of the rate at which the characters arrive. It must process the characters at an average rate which is faster than the average rate at which they can arrive, otherwise the FIFO buffer will become full and data will be lost. In other words, the buffer allows the input data to arrive in bursts, and the main program can access them when it is ready.

The following figure shows the situation for character reception.
The structure for interrupt-driven character transmission is similar, except for one minor detail which must be resolved. Output device interrupt requests come in two varieties – those that request an interrupt on the transition to the ready state, and those that request an interrupt when they are in the ready state.

**Output Device Interrupt Request on Transition to Ready**

In this case, an output device requests an interrupt when it finishes processing the current output to indicate that it is now ready for the next output. In other words, the output ISR is invoked only when the output device transitions from a “busy” condition to a “ready” condition. In the context of serial port transmission, this creates two problems:

- When the background thread puts the first byte into the FIFO buffer, the output device is idle and already in the “ready” state, so no interrupt request from the output device is about to occur. The output ISR will not be invoked and the data will not be removed from the buffer.

- If somehow started, the “interrupt – FIFO_Get – output” cycle will repeat as long as there is data in the buffer. However, if the output device ever becomes ready when the buffer is empty, no subsequent interrupt will occur to remove the next byte placed in the buffer.

In these situations, the hardware normally provides a mechanism to determine whether or not the output device is busy processing data, such as a flag in a device status register. In these cases, the main work of the ISR should be placed in a separate function (e.g. `SendData`) that actually outputs the data.

The background thread checks the output busy flag every time it writes data into the buffer. If the device is busy, then a device ready interrupt is expected and nothing needs to be done; otherwise, the background thread arms the output and calls `SendData` to “kick start” the output process.

The `SendData` routine is responsible for retrieving the data from the buffer and outputting it. If there is no more data in the buffer, then it must disarm the output to prevent further interrupts.
The flowchart given below illustrates the process.

Figure 5.6

Kick starting an interrupt-driven output routine for a device that requests interrupts on transitioning from busy to ready.
Output Device Interrupt Request on Ready

In this case, an output device sets its interrupt request flag when it is idle and ready for output (this will be the case after a reset condition, too). This means that upon initially arming the interrupt for such a device, an ISR will be invoked immediately. In the context of serial port transmission this creates two problems:

- How and when do we generate the first interrupt?
- What do we do if the device is ready but there is no data to output?

The technique to handle this type of interrupt is to modify both the `OutChar` routine and the ISR. The SCI transmit interrupt is armed after every `FIFO_Put` (if the SCI transmit interrupt were already armed, then rearming would have no effect). If the transmit FIFO is empty, then the ISR should disarm the transmit interrupt.

The flowchart given below illustrates the process.

![Flowchart](image)

Figure 5.7
Communicating between Threads

Communication between threads, without the support of an operating system, is accomplished with global variables. This leads to a problem – two (or more) threads may be trying to access and operate on the same variable at the same time. For example, in a FIFO buffer implementation for sending a byte out the serial port, the background thread (main) calls `TxFIFO_Put` to place a character into the buffer. This is a safe operation, because the byte is added to the end of the buffer. However, if the `TxFIFO` is keeping track of the number of bytes in the buffer with a global variable called `NbBytes`, then it must read, increment and write to this variable. A problem arises if a foreground thread (ISR) interrupts the background thread in the middle of the read-modify-write access to the global variable – erroneous values of `NbBytes` can result.

A critical section of code is a sequence of program instructions that must not be interrupted if erroneous operation is to be avoided. A critical section must prevent access to a global variable by more than one thread.

The solution to this simple problem is to implement access to `NbBytes` as an atomic operation. An atomic operation is one that is guaranteed to finish once it is started. In the MC9S12, all but three special instructions are atomic. Therefore, if the compiler generates the following code, then `NbBytes++` is atomic (therefore not critical) because it cannot be halted in the middle of its operation:

```assembly
inc NbBytes
```

On the other hand, if the compiler generates the following code, then `NbBytes++` is nonatomic (therefore critical) because it can start, then be interrupted:

```assembly
ldaa NbBytes
inca
staa NbBytes
```

We must study the assembly language output produced by the compiler to determine whether a section of code is critical or not.
Since we are not using a real-time operating system (which would inherently support a multithreaded program by providing interthread communication mechanisms), one way of protecting the integrity of shared global variables is to disable interrupts during the critical section. This is a simple and acceptable method of protecting a critical section for a small embedded system.

It is important not to disable interrupts too long so as not to affect the dynamic performance of other threads. Notice also that interrupts are not simply disabled then enabled, but rather the interrupt status is saved, the interrupts disabled, then the interrupt status is restored:

```
asm pshc;  // save state of interrupts
asm sei;   // disable interrupts
NbBytes++;  // critical section
asm pulc;  // restore state of interrupts
```

Consider what would happen if you simply added an “interrupt disable” at the beginning and an “interrupt enable” at the end of a subroutine, then called it with the interrupts disabled.

### Critical Sections in C

In C, macros should be defined to enter and exit critical sections of code:

```c
#define EnterCritical() { asm pshc; asm sei; asm leas 1,sp; }
#define ExitCritical() { asm leas -1,sp; asm pulc; }
```

You should understand the operation of these macros from a cursory knowledge of the MC9S12 assembly language.

The `EnterCritical()` macro firstly pushes the `CCR` register onto the stack, then sets the `I` bit to disable interrupts. The stack pointer has now been modified, unbeknownst to your compiled C code, which probably requires the stack pointer to be in its original position so as to address local variables. Therefore, the last assembly language statement in the macro increments `SP` to bring it back to its original position (before we pushed the `CCR`). We have now placed a “hidden” byte onto the stack.

The `ExitCritical()` macro firstly decrements the `SP` so that it points to the saved `CCR`, and then pulls it back into the `CCR` register. The `SP` is now in its original position (before we pushed the `CCR`) and code can continue as normal.
The problem with this approach is that the “hidden” saved CCR register that is sitting on the stack can be overwritten if the subsequent C code utilises the stack. We are therefore prohibited from making a function call (which utilises the stack for the return address and parameter passing) within our critical section of code. Worse than that though, the compiler may place its own function calls to library routines, such as 32-bit arithmetic operations, without us being explicitly aware. In these instances, what appears to be correct code will “crash” the software when the CCR register gets loaded with a corrupted value.

We should therefore restrict the usage of these macros to small portions of code where we understand the stack limitations.

A better way of implementing critical sections that preserves the stack and allows subroutines to be called from within a critical section is by declaring the following macros:

```c
#define EnterCritical() { asm tfr ccr,a; asm staa savedCCR; asm sei; }
#define ExitCritical()  { asm ldaa savedCCR; asm tfr a,ccr; }
```

You need to declare a local variable called `savedCCR` in the function that contains the critical section, before using these macros, as shown in the example below:

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

**Listing 5.2 – Creating a critical section**

Since the compiler has now reserved space for the `savedCCR` on the stack, it is safe to call functions (either implicitly or explicitly) within the critical section.
Interrupt Priority

As soon as an ISR is called, further interrupts are automatically disabled. If you wish to create a prioritised interrupt scheme, you need to re-enable interrupts within ISRs that should have a lower priority than others.

If more than one interrupt source is requesting service, the priority of service is determined by its position in the vector table. Reset has the highest priority, followed by the Clock Monitor fail reset, etc. Using the CodeWarrior interrupt numbering scheme, the lowest number has the highest priority.

You can promote one interrupt vector to be the highest priority by writing a value to the HPRIO register.
Lecture 6 – Timing Generation & Measurement


Introduction

The Enhanced Capture Timer (ECT) module has the capability of capturing events and time-stamping them, and of generating events at certain times. It also has a pulse accumulator that can be used to count external pulses without the need for software intervention. A modulus down-counter can be used to generate periodic interrupts.

Timer Module

A simplified block diagram of the timer module is shown below:

![Block Diagram of Timer Module](image)

There is a 16-bit free-running timer called TCNT. This is used to time-stamp an input event (an *input capture*) or to trigger an output event (an *output compare*). There is also a 16-bit modulus down-counter that can be used to generate periodic interrupts with much greater precision than the RTI module of the CRG block.

The “input capture / output compare” block is just a register called TCn, where n is the channel number, that gets loaded with the current value of TCNT for an
input capture event, and which holds a desired value of TCNT to trigger an output compare event.

The types of events to capture, or to initiate on a successful compare, are setup through various control registers. For inputs, it is possible to capture rising and falling edges. Outputs can be made to toggle, clear or be set.

**Modulus Down-Counter**

A block diagram of the modulus down-counter is shown below:

![A simplified modulus down-counter block diagram](image)

**Figure 6.2**

The modulus down-counter can be used as a time-base to generate a periodic interrupt. It can also be used to latch the values of the input capture registers and the pulse accumulators to their holding registers. The action of latching can be programmed to be periodic or only once.
Output Compare

Output compare can be used to create square waves, generate pulses, implement time delays, and execute periodic interrupts. You can use output compare together with input capture to measure period and frequency over a wide range and with varying resolution.

A channel set up as an output compare channel will trigger an output action when the output compare register is equal to the free-running timer. A block diagram of the output compare action is shown below:

A compare result output action can be set up using the \texttt{TCTL1} and \texttt{TCTL2} registers. The options are:

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer disconnected from output pin logic</td>
</tr>
<tr>
<td>Toggle \texttt{OCn} output line</td>
</tr>
<tr>
<td>Clear \texttt{OCn} output line to zero</td>
</tr>
<tr>
<td>Set \texttt{OCn} output line to one</td>
</tr>
</tbody>
</table>
One simple application of output compare is to create a fixed time delay. Let \( \text{delay} \) be the number of cycles you wish to wait. The steps to create the delay are:

1. Read the current 16-bit \( \text{TCNT} \).
2. Set the 16-bit output compare register to \( \text{TCNT} + \text{delay} \);
3. Clear the output compare flag.
4. Wait for the output compare flag to be set.

This method will only work for values of \( \text{delay} \) that fall between a minimum value (the time it takes to implement steps 1 to 3) and 65536. It will function properly even if \( \text{TCNT} \) rolls over from 0xffff to 0, since the 16-bit addition is really a modulo 0x10000 addition.

**Example**

The output compare feature is a convenient mechanism to create an interrupt-driven real-time clock. We will simply increment a global variable, Time, every 1 ms.

```c
const UINT16 RATE = 2000; // timer rate

void interrupt 1 TOC3_ISR(void)
{
    TFLG1 = TFLG1_C3F_MASK; // acknowledge OC3F
    TC3 = TC3 + RATE;       // executed every 1 ms
    Time++;                 // increment time
}

void TOC3_Init(void)
{
    asm sei                // ensure interrupts are disabled
    TIOS_IOS3 = 1;         // enable ch 3 as output compare
    TSCRZ = 0x03;          // set prescaler
    TIE_C3I = 1;           // arm TC3
    TC3 = TCNT + RATE;     // executed every 1 ms
    TSCR1_TEN = 1;         // enable TCNT
    asm cli                // enable interrupts
}
```

If the interrupt handler were to execute \( \text{TC3} = \text{TCNT} + \text{RATE} \); instead of \( \text{TC3} = \text{TC3} + \text{RATE} \), then interrupts would be requested at a fluctuating rate a little bit slower than every \( \text{RATE} \) cycles.
**Coupled Output Compare**

The channel 7 output compare module can be configured such that an output compare event on it will cause changes on some or all of the other output compare pins. This coupled behaviour can be used to create synchronous signals. For example, we can create pulses that start together or end together.

**Input Capture**

A channel can be set up as an *input capture* channel. We can use input capture to measure the period or pulse width of 5V CMOS signals. The input capture system can also be used to trigger interrupts on rising or falling transitions of external signals. A simplified block diagram of a channel set up for input capture is shown below:

![Input Capture Block Diagram](image)

The input capture edge detection circuits can be set up using the TCTL3 and TCTL4 registers. The options are:

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture disabled</td>
</tr>
<tr>
<td>Capture on rising edges only</td>
</tr>
<tr>
<td>Capture on falling edges only</td>
</tr>
<tr>
<td>Capture on any edge (rising or falling)</td>
</tr>
</tbody>
</table>

**Figure 6.4**

The output compare on channel 7 can be coupled to the other channels.
Two or three actions result from a capture event:

1. The current 16-bit TCNT value is copied into the input capture register, TCn.
2. The input capture flag is set in TFLG1.
3. An interrupt is requested when the mask bit is 1 in TIE.

Example

The basic idea of pulse-width measurement is to cause an input capture event on first the rising edge and then the falling edge of an input signal. The difference between these two times will be the pulse width. The resolution of the measurement is determined by the rate at which TCNT is incremented. We will use Channel 1 of the timer for the implementation.

```c
UINT16 PulseWidth;       // units of 500 ns
BOOL Done;               // true on detecting a falling edge

void interrupt 9 TIC1_ISR(void)
{
    static UINT16 rising;  // TCNT at rising edge
    if (PTT_PTT1)          // PT1=1 if rising
    {
        rising = TC1;        // record time of rising edge
        TCTL4_EDG1A = 0;     // set edge detection
        TCTL4_EDG1B = 1;     //   to falling edge only
    }
    else
    {
        // the measurement
        PulseWidth = TC1 - rising;
        Done = bTRUE;
    }
    // ack, fast flag clear is used
}

void TIC1_Init(void)
{
    asm sei                // ensure interrupts are disabled
    TIOS_IOS1 = 0;         // enable ch 1 as input capture
    DDRT DDRT1 = 0;        // PT1 is input capture
    TSCR2 = 0;             // set pre-scaler for 500 ns ticks
    TIE_C1I = 1;           // arm TC1 interrupts
    TCTL4_EDG1A = 1;       // set edge detection
    TCTL4_EDG1B = 0;       // to rising edge only
    TSCR1_TFFCA = 1;       // use fast flag clear all
    TSCR1_TEN = 1;         // enable TCNT
    TFLG1 = TFLG1_C1F_MASK; // clear CIF if set
    Done = bFALSE;         // no measurement yet
    asm cli                // enable interrupts
```
Pulse Accumulator

A pulse accumulator counts the number of active edges at the input of its channel.

The delay counter can be used to only accept pulses with a minimum duration.

There are two modes of operation for the pulse accumulators, *latch* mode and *queue* mode.

In latch mode, the value of the pulse accumulator is transferred to its holding register when the modulus down-counter reaches zero or when the force latch control bit ICLAT is written. At the same time, the pulse accumulator is cleared.

In queue mode, reads of an input capture holding register will transfer the contents of the associated pulse accumulator to its holding register. At the same time, the pulse accumulator is cleared.

Further Information

For more information on the enhanced capture timer block, consult the Freescale *ECT_16B8C User Guide* document.
Lecture 7 – The Embedded Software Tool Chain


Introduction

A software tool chain is a set of programs that you invoke in a sequential fashion to turn source code into executable machine code. In an embedded system, the end result is machine code that is “burned” into on-chip non-volatile memory, such as Flash memory. The tool chain consists of many programs such as compilers, assemblers, linkers and burners. Building software applications for embedded systems requires good knowledge of the development tools as well as the targeted hardware so that maximum design flexibility and utility is achieved.

Overview

Tool chains are vendor and hardware dependent, but a number of fundamental concepts are common to most development tool chains. Knowing about these opens the door to a wealth of information provided by these tools. All tool chains translate software from source code (e.g. a high-level language such as C) to machine code. An example tool chain is shown below:

![Figure 7.1 – An embedded software tool chain](image)

Preprocessor

The first tool, called a preprocessor, performs several mechanical operations to prepare a source file for the compiler. Such duties include macro processing, selecting source text for compilation, or incorporating different shared files into the file that will ultimately be compiled.
7.2

Compiler

The compiler is a tool for translating programs into a variety of forms. One such form is assembly language. More specifically, the compiler for an embedded system is usually a cross compiler that runs on one machine, called the development platform (e.g. a PC), and generates code for a different machine, called the target machine (e.g. an MC9S12).

Assembler

The assembler converts the collection of assembly language modules produced by the compiler into machine language or object code. Machine language represents each of the program’s instructions as a specific collection of 0’s and 1’s that the machine is designed to understand. Each machine language instruction consists of an operation code, or opcode, and normally one or more operands on which the operation should act. An assembler normally produces relocatable code, which means that all memory addresses of both code and variables are as yet unknown.

Linker

The linker is responsible for combining all of the object code into a single file and resolving addresses (or identifying address problems). All addresses are now absolute. The object code can come from user-written modules or library modules. Library modules are collections of object code supplied by vendors to provide specialized but common functionality, such as string handling, 32-bit operations (on a 16-bit device), floating-point emulation, etc.

Loader

The loader has the job of translating the absolute object code created by the linker into a form that is suitable for downloading to a burner or debugger so that the target system can be programmed. The Motorola S-record and the Intel HEX format are two output formats which have been developed with the programming of non-volatile memory in mind.
Building a Software Project

Embedded applications rely on fairly large pieces of software and therefore a modular approach to the software is generally taken. We also frequently have a large number of people involved in developing software modules. Typically, each piece of software is developed as a separate “implementation” C file (.c) with an associated “interface” header file (.h). The final program needs to bring together all the implementation files, header files, library files and other information in a process known as a “build”. The entire process is illustrated below:

Figure 7.2 – Building a C program
The Integrated Development Environment

The CodeWarrior Integrated Development Environment (IDE) encapsulates the entire process of creating the high-level source code in an editor and launching the appropriate tools in the tool chain to produce the desired output. It can also launch a debugger for the target system.

The Project Manager

The IDE gathers source, library, resource, and other files into a *project*. The Project Manager manipulates the information stored in the project. The figure below shows the Project Manager interactions with the IDE tools:

![Diagram showing Project Manager interactions]

*Figure 7.3 – Project Manager interactions*
**Build Targets**

For any given build, the project manager tracks:

- files and libraries
- link order
- dependencies
- compiler, linker, and other settings

The IDE stores this information in a *build target*. As the project changes, the project manager automatically updates the build target. The project manager also coordinates program builds, using the build-target information to call the appropriate tools in the correct order with the specified settings. For example, the project manager directs the build system to compile only those source files that rely on information in a modified file.

The project manager also supports multiple build targets within the same project file. Each build target can have its own unique settings, and even use different source and library files. For example, it is common to have both debug and release build targets in a project, as shown in the figure below:

![Figure 7.4 – Build targets in a project](image)

Note that both build targets share File #1 and File #2.
The Editor

The editor is used to create text files and source files, and is the entry-point for human development of software. The editor is used to:

- **Manage text files** – the editor includes common word-processing features for creating and editing text files. Sample text files include “Read Me” files and release notes.
- **Manage source files** – the editor includes additional features for creating and editing source files.

In addition to the usual editing functions, you can use the editor to complete these tasks:

- Open interface and header files
- Find function definitions
- Set and clear markers
- Modify file formats
- Control syntax coloring
- Execute version-control operations
- Determine a file’s save state

Editing

The editor provides these features to help edit source code:

- **Select and indent text** – the editor can select text by line, routine, or rectangular selection. The editor also handles text indentation.
- **Balance punctuation** – the editor can find matching pairs of parentheses (), brackets [], and braces {}. Most programming languages, such as C, produce syntax errors for punctuation that lacks a counterpart.
- **Complete code** – the IDE can suggest ways to complete the symbols you enter in a source file.

Navigating

The editor provides these features to help navigate source code:

- **Find specific items** – the editor finds interface files, functions, and lines of source code.
- **Go to a specific line** – the editor can scroll to a specific line of source code.
- **Use markers** – the editor allows labeling of specific items of text. These labels, or markers, provide intuitive navigation of text.
The Compiler and Assembler

The compiler and assembler operate in tandem to take high-level source code and convert it into machine-readable object code. Technically, a compiler will produce assembly language in place of the high-level language instructions, whilst the assembler simply translates the assembly language into object code. These days the compiler / assembler is usually the one program.

The compiler / assembler is used to:

- **Generate object code** – the compiler translates source code into object code. Source code is written in a high-level language such as C. Object code represents the same source instructions in a language that the computer directly understands.

- **Flag syntax errors** – the compiler highlights source code that generates syntax errors. Syntax errors result from failing to follow valid structure in a programming language. In C, a common syntax error is forgetting to end a statement with a semicolon.
7.8

The Linker

Linking is the process of assigning memory to all global objects (functions, global data, strings, and initialization data) needed for a given application and combining these objects into a format suitable for downloading to a debugger or a burner. After a successful linking session, the linker generates an absolute file (.abs) containing the target code as well as some debugging information. It also generates a file that is suitable for a burner to use in writing to non-volatile memory, such as Flash.

There are several types of linkers. They are used to complete these tasks:

- **Combine code** – the linker combines source-file object code with object code from library files and other related files. The combined code represents a complete computer program.

- **Create a binary file** – the linker processes the complete program and generates a binary file. Sample binary files include applications and shared libraries.

In CodeWarrior, the **Target Settings** panel in the **Project Settings** window contains an option to select a linker to produce a specific binary file type. The common binary file types are:

- **Applications** – applications, or executable files, are programs that perform a specific task. In an embedded environment, the application is single-purpose and custom-made for the hardware and its environment.

- **Libraries** – libraries contain code for functions that are common amongst many applications. Libraries simplify programming tasks and enhance re-usability.

- **Specialized files** – specialized files are designed for highly efficient operation in a specific context. Such files usually support a particular combination of hardware and software to perform tasks. For example, some chips in the MC9S12 family have additional hardware with microcode engines that can execute small programs in parallel with the main CPU.
The Loader

Loaders come in three varieties – debuggers, burners and bootloaders.

The Debugger

It is now common for microcontrollers to have Flash memory and a background debug module as integral parts of the chip. The debugger is responsible for loading the final memory image into the embedded hardware. This usually requires an intervening debugger hardware module which connects between the PC (running the debugger) and the embedded hardware.

The Burner

A burner is a utility that takes a special file produced by a linker and sends it to hardware designed to program non-volatile memory, such as Flash. Output from the burn process usually goes directly to a “Flash burner” connected to a USB port of the PC. In systems with external memory, the burner is responsible for splitting the application into different Flash memories for systems with 2- or 4-byte bus widths.

Burners typically take as their input one of two industry standard human-readable ASCII text file formats – either a Motorola S-Record file (.s19) or an Intel Hex file (.hex).

The Bootloader

It is now common practice for embedded hardware to contain special “bootloader” software that resides in a protected part of non-volatile memory to enable field upgrades of the firmware. The bootloader is responsible for erasing the existing firmware, upgrading to the new firmware, and executing the new firmware. Bootloaders are usually written so that they can read Motorola S-Record files or Intel Hex files that are sent over a communication link (such as a serial port / USB), since these burner files contain both address and data information.
Example

We will follow the build process through the software tool chain for a simple program to see the various input files that each tool requires and to see the output files that each tool generates.

Preprocessor

The preprocessor processes lines marked with the # symbol in the first column (these are known as pre-processor directives). As an example, when compiling `main.c` it finds a `#include "types.h"` directive and so it incorporates `types.h` into a new internal `main.c` (along with other `.h` files as specified by other directives) which will ultimately be passed to the compiler:

```
main.c

//------------------------------------------------------------------------------
// Filename: main.c
// Description: Lab 1
// Author: PMcL
// Date: 03-Apr-06

void main(void)
{
    ModCon_Setup ();
    ModCon_Init ();
    for (i;)
    {
        SCI_Poll ();
        ModCon_HandlePackets ();
    }
}

//------------------------------------------------------------------------------
// Filename: types.h
// Description: Declares new type definitions
// Author: PMcL
// Date: 03-Apr-06

typedef char INT8;
typedef int INT16;
typedef long INT32;

```

```
 Preprocessor

main.c (in memory)

//------------------------------------------------------------------------------
// Filename: main.c
// Description: Lab 1
// Author: PMcL
// Date: 03-Apr-06

void main (void);

//------------------------------------------------------------------------------
// Filename: types.h
// Description: Declares new type definitions
// Author: PMcL
// Date: 03-Apr-06

typedef char INT8;
typedef int INT16;
typedef long INT32;

```

```
```
Compiler and Assembler

The compiler now takes each module in turn and compiles the C into assembly language. The assembler then creates object code from the assembly language. The compiler and assembler operate together during the “compile” phase, and the output generated is relocatable object code (.o) for each module as well as a listing file (.lst) for each module that shows the C code interspersed with assembly language / object code:

For example, examine the following lines from main.lst:

```
 61:    ModCon_Setup();
 0000 160000 _JSR_ ModCon_Setup
```

Line 61 shows the C code, a call to function ModCon_Setup(). The next line begins with the address of the resulting object code within the current module, 0000, in hexadecimal. Following the address we have the object code itself, 160000, in hexadecimal. In this case the operation is a 3-byte “jump to
subroutine (JSR)” operation whose op-code begins with 16. The next two bytes of the instruction are for a 16-bit address, but in this case the compiler / assembler has inserted 0000 in hexadecimal. Note that the compiler / assembler does not know the address of the subroutine ModCon_Setup(), since that will be the job of the linker, so it has reserved space for it and filled it with zeros instead.

The number in brackets, in this case [4], gives the number of clock cycles that it takes to execute the instruction. For a branching instruction, you may see something like [3/1] which means it takes 3 cycles to take the branch, and 1 cycle if it doesn’t.

Lastly, the line ends with the actual assembly language instruction, in this case

```
JSR   ModCon_Setup.
```

Since the assembly language instruction took 3 bytes to store, the next address for the compiled code in the module will be 0003. You can see this just below line 62.

An *identifier* in the C language is a name given to a variable or function. During compilation, the names of all variables and functions are identified, and an entry is made in a data structure called the *symbol table*. If the compiler is able to find the definition of an identifier within the module, then information relating to its declaration or appearance in the module, such as its type, scope and sometimes its location, is stored in the symbol table. If it cannot, and it hasn’t been told that the identifier is defined in some other module (via the `extern` directive), it cites the identifier as being undefined. If the `extern` directive is present to tell the compiler to defer concerns about the definition (i.e. that it has been taken care of in another module) and the linker cannot find that definition, the process ends with a *link error*.

The location of each identifier is with respect to the start of the module where it is defined – all compiled / assembled modules are *relocatable*, which means everything is addressed with respect to the module, but the module is yet to receive an address.
Linker

The linker takes each compiled module as relocatable object code (.o or .lib) and attempts to resolve all the addresses, i.e. to place absolute addresses (or jump offsets) into the op-codes where space was previously reserved by the compiler. The linker gets information about the types and locations of memory that it can use in a parameter file (.prm). The output is a file (.abs) with absolute memory addresses and op-codes that is to be loaded into program memory. The linker also creates a human-readable format of the addresses and op-codes for use by the loader (.s19). It also creates a map file (.map) which gives detailed information on the memory allocations that it made.
The Parameter (.prm) File

The parameter file gives information about 4 different memory areas:

1. **Segments** of physical memory.
2. **Placement** of code and data.
3. The **stack size** in RAM.
4. The **vector** location for the reset condition.

Segments are physical areas of memory. Segments are defined between the keywords **SEGMENTS** and **END**. For example, the lines from `project.prm`:

```plaintext
/* EPROM */
EEProm   = READ_ONLY 0x0400 TO 0x07FF;
/* RAM */
RAM      = READ_WRITE 0x0800 TO 0x3FFF;
/* non-paged FLASHs */
ROM_C000 = READ_ONLY 0xC000 TO 0xFEFF;
```

define, respectively, non-volatile read-only memory (Flash EEPROM) from address 0x0400 through to 0x07FF, RAM from 0x0800 through to 0x3FFF, and non-paged Flash ROM from 0xC000 through to 0xFEFF. These definitions are based on the physical hardware and come straight from the Freescale data sheet for the MC9S12A512. Since the MC9S12 allows its memory to be remapped via internal write-once registers, if the MC9S12 application software changes the memory map then the linker file will need to be updated to reflect the change.

The actual placement of code and data by the linker into the previously defined memory segments occurs between the keywords **PLACEMENT** and **END**. For example, the lines from `project.prm`:

```plaintext
STARTUP,          /* startup data structures */
ROM_VAR,          /* constant variables */
STRINGS,          /* string literals */
DEFAULT_ROM,      /* runtime routines */
NON_BANKED,       /* copy down information */
COPY              /* copy down information */
INTO  ROM_C000;
SSTACK,           /* allocate stack first */
DEFAULT_RAM       INTO  RAM;
```
show the placement of all startup code, constants, strings and code into the physical segment defined as **ROM_C000**, and the stack and global variables into the physical segment defined as **RAM**. The default placements called **DEFAULT_ROM** and **DEFAULT_RAM** are pre-defined by CodeWarrior and tell the linker where to place your code and data. Code and data placements can be controlled in the C environment by the pre-processor directives **#pragma CODE_SEG** and **#pragma DATA_SEG**, respectively. This is handy if you specifically want to place code into a particular area of physical memory, such as a different non-paged Flash memory area (e.g. the one starting at 0x4000) or a paged Flash memory area.

Note that the allocation of the stack into RAM occurs before the global variables. The stack in the MC9S12 grows downwards in memory, and so the placement of the stack first in RAM ensures that when the stack reaches its limit it will not continue to “grow” into RAM where global variables reside and corrupt them (in this case it grows into EEPROM where of course it cannot cause write problems, but the software will almost certainly “crash” as the stack cannot operate in the intended manner).

The line containing **STACKSIZE 0x100** tells the linker to reserve 256 bytes of memory in RAM (in the data segment called **SSTACK**) . This is used by the CodeWarrior startup code to declare space for the stack and to initialise the MC9S12 stack pointer register (**SP**) to point to the top of the declared stack space.

The last line of the parameter file contains the statement:

```
VECTOR 0 _Startup
```

This tells the linker to insert the address of the _Startup function into the vector table at location 0, i.e. at the reset vector location. Note that the _Startup function is declared in Start12.c which is automatically created when you start a new CodeWarrior project. It is responsible for the zero-out and copy down stages of C program initialization, as well as initializing the stack and performing various other housekeeping duties before calling your **main()**.
The Absolute (.abs) File

The absolute file is not directly readable by humans or a text editor as it contains object-code of the MC9S12. However, we can see in a debugger exactly what code is executing at the assembly language level and this gives us a clue as to what the absolute file contains. We also know that the linker has had to combine all the relocatable object code into absolute object code, i.e. to resolve all addresses. From the example, we can see that the absolute file has combined all the modules and filled in all addresses and relative jump offsets.

The following lines:

C03B 16C29C       [4]     JSR   ModCon_Setup
C03E 16C1CF       [4]     JSR   ModCon_Init

when compared to the compiler/assembler output, show that the linker has inserted the relevant function addresses into the JSR instructions. We can see that the ModCon_Setup function has an absolute address of 0xC29C, and the ModCon_Init function has an absolute address of 0xC1CF. We can also see that the linker has placed the first instruction into absolute memory at 0xC03B, and since the instruction occupies 3 bytes, that the next instruction resides at 0xC03E.

Note that the start12.c code occupies the first part of the absolute file, and will therefore be at the beginning of the DEFAULT_ROM placement. Since DEFAULT_ROM is placed into the ROM_C000 segment, the first line of code in start12.c will reside at 0xC000. If you examine start12.c you will see that the Startup function calls another function, written in assembly language, called Init. Thus, Init will reside in memory at 0xC000 and Startup will reside in memory many bytes later, such as at 0xC029. This Startup address is inserted at the top of the vector table by the linker via the last line of the parameter file (as previously mentioned). Your main() program will occupy Flash memory immediately after the startup code – in this case at 0xC03B.
The Map (.map) File

The map file gives information about the memory allocated to functions, variables, hardware registers and the stack, as well as statistics and dependencies. The map file is organised into 14 parts:

Target Section
The target section lists the target processor, the memory model, the file format and the linker.

File Section
This section lists the object code (.o) files that were included by the linker.

Startup Section
The startup section tells you the entry point of the code (i.e. the code called on a reset interrupt) as well as addresses used in the “zero out” and “copy down” stages of C program initialization.

Section-Allocation Section
This section tells you where the C code and data sections (such as .text, .bss, etc.) reside in specific memory segments. The sections are classified as read only (R), read/write (R/W) and no initialization (N/I). R sections are used for code, initialization data and runtime library routines, R/W sections for variables and the stack, and N/I for hardware registers.

Vector-Allocation Section
The vector allocation section lists the vector addresses and values that are allocated by the linker (that are defined in the .prm file).

Object-Allocation Section
An object in this instance is a function or variable. This section lists the address of each object declared in each module (.o file), the size of each object, its C section (such as .text, .bss, etc.), and the number of times it is referenced.
Module Statistic
For each module (.o file), a summary of the Data, Code and Const sizes is given.

Section Use in Object-Allocation Section
This lists the objects (functions and variables) that are allocated to the various C sections (such as .text, .bss, etc.).

Object List Sorted by Address
This lists the objects (functions and variables) in physical memory address order, their size, the number of times referenced, and the C section. It is effectively a sorted version of the Object-Allocation Section.

Unused Objects Section
This lists the functions and variables that are declared but not used. Since the linker is “smart”, is does not include functions and variables into the absolute file if they are never referenced. You will generally see many library functions and a couple of library variables that are not used by your application.

Copydown Section
The copydown section is the place in Flash memory where the initial values of any initialized variables are stored. This section lists each initialized variable and its value. It gives the address of the copydown ROM location, the RAM address of the initialized variable, its size, its name, and its initial value.

Object-Dependencies Section
The object-dependencies section lists objects (functions and variables) that each function is dependent upon.

Dependency Tree
The dependency tree depicts function dependency in a graphical format. It does not indicate variable usage.

Statistics Section
The statistics section tells you the number of code blocks to be downloaded into Flash memory and the total number of bytes to be downloaded.
The **S19 (.s19)** File

The Motorola S-record format encodes binary data as ASCII hexadecimal text. It is also known as the S19 format. The format was devised to encode programs or data files in a printable format for transport between computer platforms. The format also provides for editing of the S-records and monitoring the cross-platform transfer process.

Each S-record is a character string composed of several fields. Each byte of binary data is encoded as a 2-character hexadecimal number, in Big Endian format.

The five fields that comprise an S-record are shown below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Printable Characters</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>2</td>
<td>S-record type – S0, S1, etc.</td>
</tr>
<tr>
<td>Record Length</td>
<td>2</td>
<td>Character pair count in the record, excluding the type and record length.</td>
</tr>
<tr>
<td>Address</td>
<td>4, 6 or 8</td>
<td>2-, 3-, or 4-byte address at which the data field is to be loaded into memory.</td>
</tr>
<tr>
<td>Code / Data</td>
<td>0 – 2n</td>
<td>From 0 to n bytes of executable code, memory loadable data, or descriptive information.</td>
</tr>
<tr>
<td>Checksum</td>
<td>2</td>
<td>Least significant byte of the one’s complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code / data fields.</td>
</tr>
</tbody>
</table>
The various Motorola upload, download, and other record transportation control programs, as well as cross assemblers, linkers, and other file-creating or debugging programs, utilize only those S-records that serve the purpose of the program.

The S-record types used by CodeWarrior are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Header record for each block of S records. The code / data field may contain any descriptive information identifying the following block of S-records. The address field is normally 0s.</td>
</tr>
<tr>
<td>S1</td>
<td>Record containing code / data and the 2-byte address at which the code / data is to reside.</td>
</tr>
<tr>
<td>S2-S8</td>
<td>Ignored for the MC9S12A512.</td>
</tr>
<tr>
<td>S9</td>
<td>Termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. There is no code / data field.</td>
</tr>
</tbody>
</table>

As an example, consider the following S-records:

```
S00600004844521B
S118C000FEC033FDC031270E35ED31EC3169700434FB31032655
... 
S116C98088EDB505403BB704B7053A3D30E6E605E5000022
S105FFEC02914
S9030000FC
```

With color highlighting, the fields can be identified:

```
S00600004844521B  // S0 record
S118C000FEC033FDC031270E35ED31EC3169700434FB3103263D  // S1 record
... 
S116C98088EDB505403BB704B7053A3D30E6E605E5000022
S105FFEC02914  // S9 record
S9030000FC
```
The first S-record can be decoded as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>S-Record Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>S0</td>
<td>S-record type S0, indicating a header record.</td>
</tr>
<tr>
<td>Record Length</td>
<td>06</td>
<td>Hexadecimal 06 (decimal 6), indicating six character pairs (or ASCII bytes) follow.</td>
</tr>
<tr>
<td>Address</td>
<td>0000</td>
<td>4-character, 2-byte address field; zeroes.</td>
</tr>
<tr>
<td>Code / Data</td>
<td>48 44 52</td>
<td>Descriptive information – ASCII “HDR”.</td>
</tr>
<tr>
<td>Checksum</td>
<td>1B</td>
<td>Checksum of the S0 record.</td>
</tr>
</tbody>
</table>

The second S-record can be decoded as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>S-Record Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>S1</td>
<td>S-record type S1, indicating a code / data record to be loaded / verified at a 2-byte address.</td>
</tr>
<tr>
<td>Record Length</td>
<td>18</td>
<td>Hexadecimal 18 (decimal 24), indicating 24 character pairs, representing 24 bytes of binary data, follow.</td>
</tr>
<tr>
<td>Address</td>
<td>C000</td>
<td>4-character, 2-byte address field; hexadecimal address C000 indicates location where the following data is to be loaded.</td>
</tr>
<tr>
<td>Code / Data</td>
<td>FE C0 ...</td>
<td>Bytes representing the Code / Data to load at address C000 – in this case the beginning of an assembler language op-code to load the X register with the address of the zero-out area...</td>
</tr>
<tr>
<td>Checksum</td>
<td>3D</td>
<td>Checksum of the S1 record.</td>
</tr>
</tbody>
</table>

The third last line reveals that the last line of code is to be loaded into address 0xC980.

The second last line loads the vector table – in this case just the reset vector, at location 0xFFFE, with the address of the startup code, 0xC029.
The last line is the S9 termination record, and can be decoded as:

<table>
<thead>
<tr>
<th>Field</th>
<th>S-Record Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>S9</td>
<td>S-record type S9, indicating a termination record.</td>
</tr>
<tr>
<td>Record Length</td>
<td>03</td>
<td>Hexadecimal 03, indicating three character pairs (three bytes) follow.</td>
</tr>
<tr>
<td>Address</td>
<td>0000</td>
<td>4-character, 2-byte address field; zeroes.</td>
</tr>
<tr>
<td>Code / Data</td>
<td></td>
<td>There is no code / data in an S9 record.</td>
</tr>
<tr>
<td>Checksum</td>
<td>FC</td>
<td>Checksum of the S9 record.</td>
</tr>
</tbody>
</table>

**Encoding**

Each printable ASCII character in an S-record is encoded in binary. The binary data is transmitted during a download of an S-record from a host system to the embedded system. An example of encoding for the first S1 record is shown below:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Record Length</th>
<th>Address</th>
<th>Code / Data</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>0111</td>
<td>0011</td>
<td>0011</td>
<td>0101</td>
</tr>
<tr>
<td>0111</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>0011</td>
<td>0011</td>
<td>0101</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>0011</td>
<td>0011</td>
<td>0101</td>
</tr>
<tr>
<td>0111</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>0011</td>
<td>0011</td>
<td>0101</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

```

Embedded Software 2014
Lecture 8 – Concurrent Software


Introduction

A program is a list of instructions for the computer to execute. A thread is an executing program, including the current values of the program counter, registers and variables. A thread has an execution state (such as running, ready, waiting) and a saved thread context when not running. Conceptually, each thread has its own CPU. In reality, of course, the real CPU switches back and forth from thread to thread.

By contrast, a process is a conceptual entity used when dealing with multiple programs running on a general purpose computer (such as a PC with Windows®). A process has its own virtual address space and has protected access to the CPU, other processes, files, and I/O resources.

In an embedded system, there is no need for multiple programs to be executing, since the embedded system is usually designed for a specific application. Thus, in an embedded system there is no need for the concept of a process and we will deal exclusively with threads.

The execution of the main program is called the background thread. In most embedded applications, the background thread executes a loop that never ends. This thread can be broken (execution suspended, then restarted) by foreground threads (interrupt service routines). These threads are run using a simple algorithm. The ISR of an input device is invoked when new input is available. The ISR of an output device is invoked when the output device is idle and needs more data. Last, the ISR of a periodic task is run at a regular rate. The main program runs in the remaining intervals. Most embedded applications are small in size, and static in nature, so this configuration is usually adequate.
The limitation of a single background thread comes as the size and complexity of the system grows. Projects where the software modules are loosely coupled (independent) more naturally fit a multiple background thread configuration.

A *scheduler* is a piece of software that implements multiple background threads, and forms the basis of a program known as an *operating system* (OS). Synchronization tools that allow threads to interact with each other (such as *semaphores*) are also a key feature of operating systems.

Systems that implement a thread scheduler still may employ regular I/O driven interrupts. In this way, the system supports multiple foreground threads and multiple background threads.
Threads

A thread is the execution of a software task that has its own stack and registers. Since each thread has a separate stack, its local variables are private, which means it alone has access.

Multiple threads cooperate to perform an overall function. Since threads interact for a common goal, they do share resources, such as global memory, and I/O devices.

In summary, a thread:

- is the execution of a software task
- has its own stack and registers
- has local variables which are private
- cooperates to perform an overall function
A thread can be in one of three states.

**Figure 8.3**

A thread is in the *ready* state if it is ready to run but waiting for its turn.

A thread is in the *running* state if it is currently executing. With a single instruction stream computer like the MC9S12, at most one thread can be in the run state at a time.

A thread is in the *waiting* state when it is waiting for some external event like I/O (keyboard input available, printer ready, I/O device available). If a thread communicates with other threads, then it can be waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the serial port, but another thread is currently outputting, it will wait. If a thread needs information from a FIFO (calls `FIFO_Get`), then it will wait if the FIFO is empty (because it cannot retrieve any information). On the other hand, if a thread outputs information to a FIFO (calls `FIFO_Put`), then it will wait if the FIFO is full (because it cannot save its information).
An OS may use a linked list data structure to hold the ready and waiting threads. It may create a separate waiting linked list for each reason why the thread cannot execute. For example, one waiting list for “full” during a call to \texttt{FIFO\_Put}, and one for “empty” during a call to \texttt{FIF\_Get}. In general, the OS could have one waiting list associated with each cause of waiting.

In Figure 8.4, thread 5 is running, threads 1 and 2 are ready to run, and threads 3 and 4 are waiting because a FIFO is empty.
8.6

Thread Control Blocks (TCBs)

If a thread is ready, it may be granted control of the CPU by the OS at any time. Conversely, while running, the OS may stop the thread executing and make it ready. We therefore need a way for the scheduler to save and restore the state of a thread. A *thread control block* (TCB) is used to store the information about each thread.

The TCB must contain:

1) a pointer so it can be chained into a linked list;

2) a pointer to its private stack;

3) a stack area that includes local variables

While a thread is running, it uses the actual hardware registers, CCR, D, X, Y, PC, SP. In addition to these necessary components, the TCB might also contain:

4) Thread number, type, or name;

5) Age, or how long this thread has been active;

6) Priority;

7) Resources that this thread has been granted.
The structure of a typical TCB is shown below:

![TCB Structure Diagram](image)

The running thread uses the actual registers, while the other threads have their register values saved on the stack.

**Schedulers**

A scheduler is an OS component that has responsibility for switching threads between states. A scheduler has to implement two aspects of this operation. One aspect is to save the currently running thread’s state in its TCB and to restore the state of the next thread to run (the process of changing threads). The other aspect is *when* the scheduler actually changes threads, and what it does with waiting threads.

In a preemptive scheduler, the OS interrupts each thread regardless of whether the thread is “in the middle of something important” – the OS is the sole arbiter of when the thread will actually get CPU time.

The simplest scheduling system is a round-robin scheduler – a scheduler that runs each “ready” thread for a certain amount of time in a fixed cyclic order. It does this by “hooking” into a periodic timer whose ISR performs the thread changeover function.
Example

Suppose we have three dynamically allocated threads that are executed in a round-robin fashion. Even though there are three threads, there are only two programs to run, \textit{ProgA} and \textit{ProgB}. Recall that a thread is not simply the software but the execution of the software. We will have two threads executing the same program, \textit{ProgA}.

... 

\begin{verbatim}
void main(void) {
    OS_AddThread(&ProgA);
    OS_AddThread(&ProgA);
    OS_AddThread(&ProgB);
    OS_Launch(TIMESLICE); // doesn't return
}
\end{verbatim}

A circular linked list allows the scheduler to run all three threads equally.

![Figure 8.6](image)

This example illustrates the difference between a program (e.g. \textit{ProgA} and \textit{ProgB}) and a thread (e.g. Thread 1, Thread 2 and Thread 3). Notice that Threads 1 and 2 both execute \textit{ProgA}. There are many applications where the same program is being executed multiple times.

Embedded Software 2014
Other Scheduling Algorithms

A non-preemptive (cooperative) scheduler trusts each thread to voluntarily release control on a periodic basis. Although easy to implement, because it doesn’t require interrupts, it is not appropriate for real-time systems.

A priority scheduler assigns each thread a priority number (e.g. 1 is the highest). Two or more threads can have the same priority. A priority 2 thread is run only if no priority 1 threads are ready to run. Similarly, we run a priority 3 thread only if no priority 1 or priority 2 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency (response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low-priority threads may never be run. This situation is called starvation.

Timer Overview

The Enhanced Capture Timer (ECT) unit has the ability to trigger an output event (an output compare) at certain times. There are 8 channels all using the same 16-bit free-running timer called TCNT. To generate a periodic interrupt using channel 3, we need to:

- set the TCNT rate which is controlled by TSCR2
- set TSCR1 bit 7 to a 1 to enable the timer
- set TIOS bit 3 to a 1 so that channel 3 acts as an output compare
- set TC3 to the time that the next interrupt should occur

The output compare event on channel 3 triggers an interrupt if:

- Enabled, (I=0 in the CCR)
- Armed, (C3I bit in TIE set)
- Flag set (C3F bit in TFLG1 set when TCNT equals TC3)

The output compare interrupt is acknowledged in an ISR when:

- C3F is cleared by writing a 1 to TFLG1
Example

Suppose we have three statically allocated threads that are each allowed to execute for 1 ms in a round-robin fashion. Even though there are three threads, there are only two programs to run, ProgA and ProgB. We will have two threads executing the same program, ProgA. The code for these programs is shown below.

```c
int Sub(int j)
{
    i = j + 1;
    return (i);
}

void ProgA(void)
{
    int i;
    i = 5;
    while (1)
    {
        i = Sub(i);
    }
}

void ProgB(void)
{
    int i;
    i = 6;
    while (1)
    {
        i = Sub(i);
    }
}
```

The thread control block is defined as follows:

```c
struct TCB
{
    struct TCB *next;              // link to next TCB
    unsigned char *stackPt;        // local stack pointer
    unsigned char threadId;       // 1, 2, 3, ...
    unsigned char moreStack[100]; // the stack
    unsigned char initialCCR;     // pre-load stack with CCR
    unsigned short initialD;      // pre-load stack with D
    unsigned short initialX;      // pre-load stack with X
    unsigned short initialY;      // pre-load stack with Y
    void (*initialPC)(void);      // pre-load stack with PC
};

typedef struct TCB TCBType;
typedef TCBType* TCBPtr;
```
To statically allocate the threads, we can create the following structure:

```c
TCBType sys[3] =
{
    {&sys[1],       // pointer to next TCB
     &sys[0].initialCCR // initial SP
     1,               // thread ID
     {0},            // empty stack
     0x40, 0, 0, 0,   // CCR, D, X, Y
     ProgA           // initial PC
    },
    {&sys[2],       // pointer to next TCB
     &sys[1].initialCCR // initial SP
     2,               // thread ID
     {0},            // empty stack
     0x40, 0, 0, 0,   // CCR, D, X, Y
     ProgA           // initial PC
    },
    {&sys[0],       // pointer to next TCB
     &sys[2].initialCCR // initial SP
     3,               // thread ID
     {0},            // empty stack
     0x40, 0, 0, 0,   // CCR, D, X, Y
     ProgB           // initial PC
    }
};
```

Even though the threads have not yet been allowed to run, they are created with an initial stack area that “looks like” it had been previously suspended by the OS. Notice that the initial value loaded into the CCR (0x40) when the thread runs for the first time has XIRQ disabled (X=1) and IRQ enabled (I=0). When the thread is launched for the first time, it will execute the program specified by the value in the initialPC location.
The round-robin scheduler simply switches to a new thread every 1 ms:

```c
TCBPtr RunPt; // Pointer to current thread

void interrupt 11 ThreadSwitchISR(void)
{
    TFLG1 = TFLG1_C3F_MASK; // acknowledge timer interrupt
    asm ldx RunPt // point to current thread
    asm sts 2,x // save stackPt
    RunPt = RunPt->Next; // get the next thread pointer
    asm ldx RunPt // point to next thread
    asm lds 2,x // new SP
    TC3 = TC3 + 8000; // run for 1 ms
}

void main(void)
{
    RunPt = &sys[0]; // specify first thread
    asm sei // ensure interrupts are disabled
    TIOS_IOS3 = 1; // enable ch 3 as output compare
    TSCR1_TEN = 1; // enable TCNT
    TIE_C3I = 1; // arm TC3
    TC3 = TCNT + 8000; // thread switch in 1 ms
    asm ldx RunPt // point to the first thread
    asm lds 2,x // initialise the real SP
    asm rti // change threads
}
```

Listing 8.1 – Round-robin Scheduler

Note that the scheduler does not have to explicitly enable interrupts (`asm cli`) since this will happen automatically when the thread gets loaded by the `asm rti`.

---

Embedded Software 2014
Operating Systems

An operating system (OS), at the very least, provides scheduling and synchronization tools for threads. One of the most important synchronization tools provided by an OS is the semaphore.

The Semaphore

A *semaphore* is a non-negative integer, which may only be operated on by the primitive operations *wait* and *signal*. These *wait* and *signal* operations are *indivisible*. Indivisibility implies that only one thread can access each of these primitives at any one time. That is, only one primitive can modify the value of some semaphore, say $s$. The primitives can’t be interrupted or logically cut into any smaller pieces.

The primitive operations are defined as follows:

\[
\text{wait}(s) \quad \text{Decrease (indivisibly) the value of } s \text{ by 1}
\]

\[
\text{signal}(s) \quad \text{Increase (indivisibly) the value of } s \text{ by 1}
\]

Note: A semaphore $s$ may only be a non-negative integer, so that if the *wait*(s) *cannot be completed*, then a thread will be put into the waiting state. Conversely a *signal*(s) may cause a waiting thread to be made active.

Every *signal*(s) increments $s$ but a *wait*(s) is only completed if $s > 0$. 
8.14

Mutual Exclusion with Semaphores

Listing 8.2 shows how semaphores can be used to guarantee that a thread will have *uninterrupted* access to its critical code section. That is, there is mutual exclusion of other threads.

```c
void p1(void)
{
    while (1)
    {
        ...
        OS_Wait(&Mutex);
        // critical code of p1
        ...
        OS_Signal(&Mutex);
        // remainder of p1
        ...
    }
}

void p2(void)
{
    while (1)
    {
        ...
        OS_Wait(&Mutex);
        // critical code of p2
        ...
        OS_Signal(&Mutex);
        // remainder of p2
        ...
    }
}

int Mutex; // a binary semaphore

void main(void)
{
    // Mutually excluded threads
    Mutex = 1;
    OS_AddThread(&p1);
    OS_AddThread(&p2);
    OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 8.2 – Mutual exclusion using Semaphores

There are two separate functions p1 and p2, each with a critical section of code. Each of these critical sections are protected between `OS_Wait` and `OS_Signal` operations. A semaphore `Mutex` is initialised to 1 at the beginning of the main program. This value guarantees only one thread can be in its critical section at one time.
Since the speed of each thread is indeterminate we have no way of knowing which one will try to execute an `OS_Wait(&Mutex)` first. Let us assume that p1 tries first. Since the semaphore has a value of 1 at this point, the `OS_Wait` will complete (that is, `OS_Wait` will decrement Mutex to 0) and p1 will enter it's critical section.

If while p1 is in the critical section p2 attempts a `OS_Wait(&Mutex)` then the `OS_Wait` will not complete and p2 will be waiting (internally to the operating system, p2 will be added to a queue of threads that are all waiting on the semaphore Mutex).

At some later time p1 will complete it's `OS_Signal(&Mutex)` operation, and p2 may now complete it's `OS_Wait(&Mutex)` and enter it's critical section. After each thread completes it's `OS_Signal` operation, the value of the semaphore again becomes 1 allowing either thread to again gain mutually exclusive access to its critical section.
Synchronisation using Semaphores

Listing 8.3 shows how two threads p1 and p2 can synchronise their operations with each other.

```c
void p1(void)
{
    while (1)
    {
        // some amount of code
        ...
        OS_Wait(&Proceed);
        // remainder of p1
        ...
    }
}

void p2(void)
{
    while (1)
    {
        // some other amount of code
        ...
        OS_Signal(&Proceed);
        // remainder of p2
        ...
    }
}

int Proceed; // a binary semaphore

void main(void)
{
    // Synchronized threads
    Proceed = 0;
    OS_AddThread(&p1);
    OS_AddThread(&p2);

    OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 8.3 – Synchronization using Semaphores

In this example p1 will pause until p2 executes a OS_Signal(&Proceed) before it continues. Of course if p2 executes the OS_Signal before p1 can execute the OS_Wait operation, then p1 will not be held up.

This is known as asymmetric synchronisation – can you re-design this example so that symmetrical synchronisation between the two threads results?
The Producer / Consumer Problem using Semaphores

A classical problem in concurrent programming is the producer / consumer problem. Here two threads communicate through a buffer. The buffer has a finite amount of space and the producer, at its own speed, produces items and deposits them in this buffer of size $\text{SpaceAvailable}$.

The consumer, at its own speed, removes items from the buffer. Of course the consumer cannot extract items from an empty buffer nor can the producer deposit items into a full buffer.

There are to be three semaphores:

- $\text{SpaceAvailable}$: This has an initial value of the size of the empty buffer.
- $\text{ItemsAvailable}$: This has a value equal to the items in the buffer at initialisation (that is, 0).
- $\text{BufferAccess}$: This controls access to the buffer so that only one thread, producer or consumer, can gain access at one time.

Synchronization is achieved through the semaphores $\text{SpaceAvailable}$ and $\text{ItemsAvailable}$ whose initial values are the size of the buffer 40 and 0 respectively. Mutual exclusion of threads accessing the buffer simultaneously is effected by the semaphore $\text{BufferAccess}$ with initial value 1.
```c
void Producer(void)
{
    while (1)
    {
        // produce item
        ...
        OS_Wait(&SpaceAvailable);
        OS_Wait(&BufferAccess);
        // deposit item in buffer
        ...
        OS_Signal(&BufferAccess);
        OS_Signal(&ItemsAvailable);
    }
}

void Consumer(void)
{
    while (1)
    {
        OS_Wait(&ItemsAvailable);
        OS_Wait(&BufferAccess);
        // extract item from buffer
        ...
        OS_Signal(&BufferAccess);
        OS_Signal(&SpaceAvailable);
        // consume item
        ...
    }
}

void main(void)
{
    int SpaceAvailable = 40; // size of the buffer
    int ItemsAvailable = 0;
    int BufferAccess = 1;

    // producer and consumer threads
    OS_AddThread(&Producer);
    OS_AddThread(&Consumer);

    OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 8.4 – Producer / Consumer problem using Semaphores
9.1

Lecture 9 – Interfacing

*Input switches and keyboards. Analog to digital conversion. Digital to analog conversion.*

**Introduction**

An embedded system is normally designed to interact with the external world. They sometimes need to provide a human-machine interface for simple input/output operations. They also may need to measure analog quantities and output analog quantities. The following sections look at various techniques of interfacing to our microcontroller.

**Input Switches and Keyboards**

**Interfacing a Switch to the Microcontroller**

Several simple switch interfaces are shown below:

![Simple switch interfaces](image)

*Figure 9.1*

In Figure 9.1 (a), a pull-up resistor is used to convert the mechanical signal into an electric signal. When the switch is open, the input port is pulled to +5 V. When the switch is closed, the input port is forced to 0V.

In digital logic that uses TTL we usually pull-up to +5 V rather than pulling down to zero because of the current limitations on TTL inputs. CMOS digital logic can use either pull-up or pull-down. Figure 9.1 (b) shows a pull-down circuit. When the switch in this circuit is open, the input is pulled to 0 V. When
the switch is closed, the output is forced to +5 V. Notice the polarity of the switch is reversed in the pull-down interface as compared to the pull-up case.

Port J on the MC9S12 supports both internal pull-ups and pull-downs. That is, either of the first two circuits in Figure 9.1 could be implemented on the MC9S12 without the resistor, as shown in Figure 9.1 (c).

The software initialization for using Port J sets bits in the \texttt{PERJ} register to enable pull-up or pull-down. For each Port J pin that is enabled for pull-up or pull-down, the corresponding bit in the \texttt{PPSJ} register determines if it is pull-up (0) or pull-down (1). It is good programming practice to first set the \texttt{PPSJ} register, then set the \texttt{PERJ} register so that temporary glitches are avoided.

**Example**

Suppose we wish to initialize Port J for the circuit shown in Figure 9.1 (c). The software below will initialize Port J with the appropriate pull-up and pull-down.

```c
// Port J Bit 1 is connected to a switch to +5 V
// and uses internal pull-down
// Port J Bit 0 is connected to a switch to 0 V
// and uses internal pull-up

void Initialization(void)
{
    // Port J Bit 0
    DDRJ_DDRJ0 = 0;        // PJ0 is an input
    PPSJ_PPSJ0 = 0;        // pull-up on PJ0
    PERJ_PERJ0 = 1;        // pull enabled on PJ0
    PIEJ_PIEJ0 = 0;        // disarm key wake-up interrupt on PJ0
    PIFJ = PIFJ_PIFJ0_MASK; // clear key wake-up flag on PJ0

    // Port J Bit 1
    DDRJ_DDRJ1 = 0;        // PJ1 is an input
    PPSJ_PPSJ1 = 1;        // pull-down on PJ1
    PERJ_PERJ1 = 1;        // pull enabled on PJ1
    PIEJ_PIEJ1 = 0;        // disarm key wake-up interrupt on PJ1
    PIFJ = PIFJ_PIFJ1_MASK; // clear key wake-up flag on PJ1
}
```
Hardware Debouncing Using a Capacitor

Most inexpensive switches mechanically “bounce” when touched and when released. Typical bounce times range from 1 ms to 25 ms. Ideally, the switch resistance is zero (actually about $0.1\,\Omega$) when closed and infinite when open. This gives rise to the following switch timing:

![Switch timing showing bounce on touch and release](image)

**Figure 9.2**

Most keyboard devices use inexpensive switches that bounce. Hence, the electrical output “bounces” when using inexpensive switches and circuits having just a pull-up or pull-down resistor. It may or may not be important to debounce the switch. For example, if we are entering data via a keyboard, then we want to record only individual key presses. On the other hand, if the switch position specifies some static condition, and the operator sets the switch before turning on the microcontroller, then debouncing is not necessary.
A hardware method to debounce a switch places a capacitor across the switch to limit the rise time, followed by an inverter with hysteresis. With this circuit there is a significant delay from the release of the switch until the fall of the output.

![Diagram of a hardware circuit to remove switch bounce](image)

**Figure 9.3**

If the input switch is closed, its resistance will be about 0.1Ω, and the output of the 74HC14 will be high (logic 1). If the input switch is open, its resistance will be infinite, and the output of the 74HC14 will be low (logic 0). The 22 Ω is used to limit the discharge current when the switch is pressed (which causes sparks that produce carbon deposits to build up until the switch no longer works).
The touch timing with and without the capacitor is shown below:

![Diagram of touch timing with and without a capacitor](image)

**Figure 9.4**

Notice that there is minimal delay between the touching of the switch and the transition of the Schmitt inverter output. This is because the capacitor is quickly discharged through the \( 22 \Omega \) resistor.

*With a capacitor-based debounced switch, there is minimal delay between the closing of the switch and the rising edge at the microcontroller input.*

(9.1)
The voltage rise during a bounce interval when the switch is open is given by:

\[ v(t) = V_{OH} \left(1 - e^{-t/RC}\right) \]  

(9.2)

The capacitor is chosen such that the input voltage does not exceed the input high threshold voltage of the Schmitt trigger during the bouncing.

**Example**

In the example, \( R = 1\, \text{k}\Omega \), and the bounce time is \( \Delta t = 5\, \text{ms} \).

We choose \( C \) so that the voltage rise doesn’t pass the Schmitt trigger input high threshold of \( V_{T+} = 2.5\, \text{V} \) until \( 5\, \text{ms} \) has passed:

\[
V_{T+} \geq V_{OH} \left(1 - e^{-\Delta t/RC}\right) \\
1 - \frac{V_{T+}}{V_{OH}} \leq e^{-\Delta t/RC} \\
\ln\left(1 - \frac{V_{T+}}{V_{OH}}\right) \leq \frac{-\Delta t}{RC} \\
C \geq \frac{-\Delta t}{R \ln\left(1 - \frac{V_{T+}}{V_{OH}}\right)} \\
\geq \frac{-5 \times 10^{-3}}{1 \times 10^{-3} \ln\left(1 - \frac{2.5}{5}\right)} \\
\geq 7.213\, \mu\text{F}
\]

Therefore, choose \( C = 10\, \mu\text{F} \).
The release timing with and without the capacitor is shown below:

![Diagram showing release timing with and without capacitor](image)

**Figure 9.5**

There is a significant delay from the release of the switch until the fall of the output, since the capacitor charges up slowly through the $1\,\text{k}\Omega$ resistor.

**With a capacitor-based debounced switch, there is a large delay between the opening of the switch and the falling edge at the microcontroller input.**

(9.3)
Hysteresis is required on the inverter logic gate because the capacitor causes the “logic” input to rise very slowly. Thus, while the input voltage is in the transition region between “low” and “high”, a regular logic gate will be operating in its linear region, and the output will be undefined. Furthermore, any noise on the input whilst in the transition region would cause a regular gate to toggle with the noise. The hysteresis removes the extra transitions that might occur with a regular gate:

![Figure 9.6](image)

Figure 9.6
Software Debouncing

It is less expensive to remove switch bounce using software methods. It is appropriate to use a software approach because the software is fast compared to the bounce time. Typically we use a pull-up resistor to convert the switch position into a TTL-level digital signal.

![A switch interface for software debouncing](image)

**Figure 9.7**

There are several ways to implement software debouncing. In the examples below, it is assumed that the switch bounce is less than 10 ms.
Example

In this example, the microcontroller is dedicated to the interface and does not perform any other functions while the routines are running. The routine waits for the switch to be pressed (PT3 low) and returns 10 ms after the switch is pressed.

```c
void WaitPress(void)
{
    // loop here until switch is pressed
    while (PTT_PTT3 == 1);
    TC5 = TCNT + 20000;    // 10 ms delay
    // wait for switch to stop bouncing
    while (TFLG1_C5F == 0);
}

void WaitRelease(void)
{
    // loop here until switch is released
    while (PTT_PTT3 == 0);
    TC5 = TCNT + 20000;    // 10 ms delay
    // wait for switch to stop bouncing
    while (TFLG1_C5F == 0);
}

void Init(void)
{
    TIOS_IOS5 = 1;         // enable ch 5 as output compare
    DDRT_DDRT3 = 0;        // PT3 is input capture
    TSCR1_TFFCA = 1;       // use fast flag clear all
    TSCR2 = 2;             // set pre-scaler for 500 ns ticks
    TSCR1_TEN = 1;         // enable TCNT
}
```
Example

In this example, the microcontroller reads the current value of the switch. If the switch is currently bouncing, it will wait for stability.

A return value of 0 means pressed ($PT3 = 0$), and 1 means not pressed ($PT3 = 1$). Notice that the software always waits in a “do nothing” loop for 10 ms. This inefficiency can be eliminated by placing the switch I/O in a foreground interrupt-driven thread.
With a software-based debounced switch, the signal arrives at the microcontroller input without delay, but software delays may occur at either touch or release. (9.4)

Input capture is a convenient mechanism to detect changes on the digital signal. The input capture can be configured to interrupt either on the rise, the fall or both the rise and fall. Because of the bounce, any of these modes will generate an interrupt request when the key is touched or released. A combination of input capture and output compare interrupts allows the switch interface to be performed in the background.
Example

This example simply counts the number of times the switch is pressed. The IC3 interrupt occurs immediately after the switch is pressed and released. Because the IC3 handler disarms itself, the bounce will not cause additional interrupts. The OC5 interrupt occurs 10 ms after the switch is pressed and 10 ms after the switch is released. At this time the switch position is stable (no bounce).

The first IC3 interrupt occurs when the switch is first touched. The first OC5 interrupt occurs 10 ms later. At this time the global variable Count is incremented. The second IC3 interrupt occurs when the switch is released. The second OC5 interrupt does not increment the count but simply rearms the input capture system. The initialization routine initializes the system with IC3 armed and OC5 disarmed.
// counts the number of button pushes
// signal connected to IC3 = PTT3

UINT16 Count;          // times pushed
const UINT16 WAIT = 20000; // bounce wait (cycles)

void interrupt 13 TOC5_ISR(void)
{
    TIE_C3I = 1;          // arm IC3
    TIE_C5I = 0;          // disarm OC5
    TFLG1 = TFLG1_C3F_MASK; // clear IC3F
    if (PTT_PTT3 == 0)
    {
        Count++;
    }
}

void interrupt 11 TIC3_ISR(void)
{
    TIE_C3I = 0;          // disarm IC3
    TIE_C5I = 1;          // arm OC5
    TC5 = TCNT + WAIT;    // wait for 10 ms, clear C5F
}

void Init(void)
{
    asm sei                 // ensure interrupts are disabled
    TIOS_IOS3 = 0;          // enable ch 3 as input capture
    TIOS_IOS5 = 1;          // enable ch 5 as output compare
    DDRT_DDRT3 = 0;         // PT3 is input capture
    TSCR1_TFFCA = 1;        // use fast flag clear all
    TSCR2 = 2;              // set pre-scaler for 500 ns ticks
    TIE_C3I = 1;            // arm IC3
    TIE_C5I = 0;            // disarm OC5
    TFLG1 = TFLG1_C3F_MASK; // clear C3F
    TCTL4_EDG3A = 1;        // set edge detection
    TCTL4_EDG3B = 1;        // to any edge
    TSCR1_TEN = 1;          // enable TCNT
    Count = 0;
    asm cli                 // enable interrupts
}
Example

The latency of the previous example is defined as the time when the switch is touched until the time when the count is incremented. Because of the delay introduced by the OC5 interrupt, the latency is 10 ms. If we assume the switch is not bouncing (currently being touched or released) at the time of the initialization, we can reduce this latency to less than 50 μs by introducing a global Boolean variable called LastState. If LastState is true, then the switch is currently not pressed and the software is searching for a touch. If LastState is false, then the switch is currently pressed and the software is searching for a release.

```c
// counts the number of button pushes
// signal connected to IC3 = PTT3
const UINT16 WAIT = 20000; // bounce wait (cycles)

UINT16 Count;          // times pushed
BOOL LastState;        // looking for touch?
// true means open, looking for a touch
// false means closed, looking for release

void interrupt 13 TOC5_ISR(void)
{
    TIE_C3I = 1;            // arm IC3
    TIE_C5I = 0;            // disarm OC5
    TFLG1 = TFLG1_C3F_MASK; // clear IC3F
}

void interrupt 11 TIC3_ISR(void)
{
    if (LastState)
    {
        Count++;              // a touch has occurred
        LastState = bFALSE;
    }
    else
    {
        LastState = bTRUE;    // release occurred
        TIE_C3I = 0;           // disarm IC3
        TIE_C5I = 1;           // arm OC5
        TC5 = TCNT + WAIT;     // wait for 10 ms, clear C5F
    }
}
```
void Init(void)
{
    asm sei                 // ensure interrupts are disabled
    TIOS_IOS3 = 0;          // enable ch 3 as input capture
    TIOS_IOS5 = 1;          // enable ch 5 as output compare
    DDRT_DDRT3 = 0;         // PT3 is input capture
    TSCR1_TFFCA = 1;        // use fast flag clear all
    TSCR2 = 2;              // set pre-scaler for 500 ns ticks
    TIE_C3I = 1;            // arm IC3
    TIE_C5I = 0;            // disarm OC5
    TFLG1 = TFLG1_C3F_MASK; // set edge detection
    TCTL4_EDG3A = 1;        // to any edge
    TSCR1_TEN = 1;          // enable TCNT
    Count = 0;
    LastState = PTT_PTT3;
    asm cli                 // enable interrupts
}

Now the latency is simply the time required for the microcontroller to recognize and process the input capture interrupt. Assuming there are no other interrupts, this time is less than 20 cycles.
Analog to Digital Conversion

An analog-to-digital (ATD) converter (or ADC) is used to quantize an external analog signal so as to represent it digitally. If the samples of an analog signal are taken at a sufficiently high rate, then the samples furnish enough information for the analog signal to be reconstructed exactly. Once the analog signal has been converted to a digital form, it can be filtered, manipulated, and processed. The processed signal can then be converted back to an analog signal through the use of a digital-to-analog converter (DAC).

ATD

The Analog To Digital (ATD) module has the capability of sampling sixteen analog channels with either an 8-bit or 10-bit quantization. The result can also be left- and right-justified to a 16-bit word, and it can be unsigned or signed (2’s complement). The signed notation is useful when converting shifted bipolar inputs (a shifted bipolar input will have its “zero” at exactly half way between the minimum and maximum conversion values). The ATD can also be set up to automatically sample a sequence of analog channels.

A simplified block diagram of the ATD module is shown below:

The analog multiplexer (MUX) is just an analog switch that connects one of the analog channels to the sample-and-hold (S/H) block.
The sample-and-hold circuit consists of a sample capacitor and a buffer. It is important to take into account the characteristics of the S/H block in the design of the analog interface hardware external to the microcontroller. The external hardware’s output resistance and the sample capacitor form a first-order lowpass filter, also known as a single time constant (STC) circuit, or just a lowpass $RC$ circuit. This first-order filter determines the amount of time that is needed to charge the sample capacitor to a voltage that is almost equal to the true analog voltage.

The voltage held on the sample capacitor is fed into the analog-to-digital converter. There are many types of ADC – the type used in the MC9S12 series of microcontrollers is a successive approximation architecture. It functions by comparing the stored analog sample voltage with a series of digitally generated analog voltages. By following a binary search algorithm, the ADC locates the approximating voltage that is nearest to the sampled voltage.

The results of the analog-to-digital conversions are stored in separate result registers. The completion of an analog-to-digital conversion can be used to trigger an interrupt.

**Further Information**

For more information on the analog-to-digital module, consult the Freescale *ATD_10B8C Block User Guide* document.
Digital to Analog Conversion

A simple DAC can be made from a pulse width modulated (PWM) waveform. If a PWM waveform is passed through a lowpass filter, and the PWM has a sufficiently high frequency, then the output of the filter will be a smooth analog waveform corresponding to the average value of the PWM taken over many periods.

Pulse Width Modulator

A pulse width modulator is a device which varies the duty cycle (“on time” versus “total time”, in percent) of a square wave. They can be used to turn transistors on and off in an external circuit to drive devices such as DC motors and 3-phase AC motors. They can also be used to create a simple digital-to-analog converter.

A simplified block diagram of the PWM module is shown below:

There are 8 PWM channels. Each channel has an 8-bit counter called $\text{PWM}^{\text{CNT}}_n$. The counter compares to two registers: a duty register called $\text{PWM}^{\text{DTY}}_n$; and a period register called $\text{PWM}^{\text{PER}}_n$. In its simplest mode of operation, the output is set high when the counter equals the period register, and the output is set low when the counter equals the duty register.
Various constant duty cycle waveforms are shown below:

<table>
<thead>
<tr>
<th>Duty Cycle</th>
<th>PWMDTY</th>
<th>PWMPER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>25%</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>50%</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>75%</td>
<td>192</td>
<td>256</td>
</tr>
<tr>
<td>100%</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

**Figure 9.11**

The generation of fixed duty cycle square waves is only one application of the PWM module. It is more generally used to *modulate* the pulse width of the square wave. Such a waveform is shown below:

**Figure 9.12**

Such a PWM waveform can be used to generate a continually varying analog signal. The area under the PWM waveform approximates the area under the desired analog waveform. The PWM waveform needs to be filtered by external hardware to remove the high frequency components, and to leave just the fundamental and DC components. You can think of the analog output as responding to the *average* value of the PWM waveform (over a small time interval).
An external filter is usually not required when the device being driven provides an inherent filtering function. For example, a DC motor, which exhibits mechanical inertia, cannot respond to the rapid fluctuations of the PWM waveform – but it can respond to the slowly varying “average” value of the waveform. In this case, the DC motor speed would be seen to vary sinusoidally. There may also be an audible “hum” or “buzz” due to the high frequency components being within the range of human hearing (20 Hz – 20 kHz). If such a hum is undesirable, then the designer can increase the frequency of the PWM wave so that the high frequency components are out of audio range.

If an analog voltage is required, a simple buffered $RC$ circuit can be used:

![Diagram of an RC circuit](image)

**Figure 9.13**

**Further Information**

For more information on the enhanced capture timer block, consult the Freescale *PWM_8B8C Block User Guide* document.
Lecture 10 – Fixed-Point Processing


Introduction

Most microprocessors are fixed-point devices – they only have support for arithmetic with integers. Desktop PCs are relatively special in that the Intel processor (Pentium, Core 2 Duo, etc) has hardware that directly supports floating-point numbers – this is why they are fast, and this is why they are expensive. A large proportion of the die area and power consumption is taken up by a floating-point unit (FPU). Floating-point operations can be emulated in software, but the resulting overhead results in programs that run 40-100 times slower than a fixed-point program.

Therefore, when speed (i.e. time) is of primary importance in a design, it is necessary to perform arithmetic operations using fixed-point numbers. We therefore need to examine processing techniques that use integers but provide an interpretation of the resulting numbers as having fractional parts.

Q Notation

Fixed-point calculations are capable of performing fractional mathematics if an implied binary point is used in the interpretation of the integer used to represent a fractional quantity. In accordance with accepted digital signal processing (DSP) notation, we use what is called “Q notation”. The “Q” stands for quotient, or a number with a fractional part.

Since we are using a 16-bit processor, all quantities will be assumed to use either 16 bits or 32 bits for their representation. To express a fractional part, an implied binary point is required for each quantity. It is up to us as designers to keep track of these implied binary points throughout any and all calculations. For each quantity, we express its fractional part with the notation $mQn$ where $m$ and $n$ are numbers ranging from 0 to 16 for 16-bit quantities or 0-32 for 32-bit quantities. The $m$ tells how many bits are used in total. The $n$ tells how many bits are to the right of the implied binary point.
Just like a decimal point, a binary point interprets digits to the right of it as being negative powers of the base. A comparison of a decimal number and its equivalent binary number is given below:

<table>
<thead>
<tr>
<th>Decimal number</th>
<th>Equivalent binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.625</td>
<td>101.101</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
5.625 & = 5 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3} \\
& = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}
\end{align*}
\]

Figure 10.1

For example, a 6Q3 number implies 3 bits to the right of the implied binary point. A mapping of the CPU’s integer values to quantities that we interpret is made as follows:

<table>
<thead>
<tr>
<th>Register value</th>
<th>Interpreted as a 6Q0 number</th>
<th>Interpreted as a 6Q3 number</th>
</tr>
</thead>
<tbody>
<tr>
<td>101101</td>
<td>45 / 2^0 = 45 / 1 = 45</td>
<td>45 / 2^3 = 45 / 8 = 5.625</td>
</tr>
</tbody>
</table>

Figure 10.2

From this, it should be apparent that to interpret a register value as a \( mQn \) value, we simply divide the raw value by \( 2^n \). To store a fractional number in \( mQn \) notation, we multiply it by \( 2^n \) and truncate or round the answer to an integer. This inherent round-off error cannot be prevented.
For example, if we wished to store the number 5.628 in 16Q3 notation, we get:

\[ 5.628 \times 2^3 = 5.628 \times 8 = 45.024 \]  \hspace{1cm} (10.1)

\[ \therefore \text{store as 45} \]

In this case it is impossible to distinguish between 5.625 and 5.628 in 16Q3 notation.

The resolution of \( mQ_n \) numbers can therefore be expressed as \( 2^{-n} \). For example, in 16Q3 notation the resolution of the stored numbers is \( 2^{-3} = 0.125 \). Every number in Q3 notation will be a multiple of 0.125. Clearly it is desirable to have a large \( n \) to store fractional values with the greatest accuracy. It is in fact impossible to store 5.628 exactly (no round-off error). The best we can do using 32-bits is to store the integer part (5 in 5.628) using the least amount of bits (3 in this case) and use the rest for the fractional part. We therefore would use a 32Q29 number:

\[ \text{Interpretated as a Q29 number= } 3021509492 \ / \ 2^{29} = 3021509492 \ / \ 536870912 = 5.6279999999 \]

**Figure 10.3**
The reason we can’t store this number exactly is because when we multiply 5.628 by successive powers of two to obtain an integer, the last digits form a cyclic pattern, that will never reach a multiple of 10:

\[
\begin{align*}
5.628 \times 2 &= 11.256 \\
11.256 \times 2 &= 22.512 \\
22.512 \times 2 &= 45.024 \\
45.024 \times 2 &= 90.048 \\
90.048 \times 2 &= 180.096 \\
180.096 \times 2 &= 360.192 \\
\end{align*}
\]

(10.2)

This shouldn’t really worry us, because a 32Q29 number has a resolution of \(2^{-29} = 1.8626451 \times 10^{-9}\). The error in storing the above number as shown is therefore less than 0.00000003 %.

As an aside, we should not forget that using floating-point numbers does not increase our accuracy. Accuracy is determined purely by the number of bits, not in the way the number is stored. It shocks some people to find that floating-point units cannot store the number 0.1, precisely because of the problem stated above. However, the floating-point number can get very close to 0.1 in the same way that we can get very close to 5.628.

**Other Notations**

The Q notation is convenient because it expresses a number as powers of two. It will be shown later that this provides an efficient method to convert numbers from one Q representation to another.

We can also express numbers using a base other than 2. For example, suppose we say that the number 1000 is to be interpreted as 1. We say that the number has 1000 as a base, or unity value, and that 1000 = 1 per unit (p.u.). The number 5.628 in this method would be represented as 5628, which is exact. Why don’t we use this method over Q notation? The answer is because other numbers can now not be represented exactly. Remember – the fundamental limit in accuracy is set by the number of bits, and not how they are interpreted.
Complications arise in calculations involving multiplications and divisions. For example, multiplying two numbers with a base of 1000 produces a number whose base is 1000000. To *normalise* this result back to 1000, the result would have to be divided by 1000 – this division is expensive in terms of CPU time and is to be avoided.

It should be noted that Q notation is just representing numbers with bases that are multiples of two. For example a 16Q3 number is a number with a base or p.u. value of 8.

**Fixed-Point Calculations**

Multiplying two numbers together changes the base or “per unit” value. For example, consider the following multiplication:

\[
\begin{array}{c}
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

**Figure 10.4**

Two things happen – 1) the length of the result is equal to the sum of the lengths of the two multiplicands and 2) the Q notation of the result is equal to the sum of the individual Q notations.

We can state this formally as follows:

\[
mQn \times iQj = (m + i)Q(n + j)
\]

(10.3)

A 16-bit CPU automatically handles the increase of the result length – two 16-bit operands will give a 32-bit result for multiplication. We can’t multiply the result by another number, because that would involve a 32-bit x 16-bit multiplication which is not directly supported by a 16-bit CPU. We have to emulate what a floating-point unit would do – *normalise*. This means the 32-bit
result must be converted back to a 16-bit number that has some arbitrary Q
notation. In the example above, if we wished to convert the result from a 32Q5
number (base 32) to a 16Q3 (base 8) number, we shift it right 2 bits (divide by
4 which is the amount the base has changed). We should note that in shifting,
we inevitably lose accuracy. This is the price paid for maintaining successive
calculation results within 16-bits.

We can see now why Q notation is efficient – normalisation is carried out by
shifts which are very quick in terms of CPU time (much quicker than a divide –
1 cycle time for each shift, compared with 12 cycles for a divide on the
MC9S12).

For division, we similarly have:

\[ mQn \div iQj = (m - i)Q(n - j), \quad n > j \] (10.4)

For example, a 32Q16 number divided by a 16Q8 number results in a 16Q8
quotient (if there is no overflow, i.e. the quotient fits into 16 bits).

Additions must be performed with numbers of the same Q notation. If they are
different, then normalisation to the larger base is required. For example:

```
 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 0
 0 1 0 1 0 1 0 0 1 0 0 1 0 0 0

= 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0
```

Figure 10.5

Similarly, subtraction requires normalisation of the bases so that the larger base
is common.
Example

We will develop the equations that MC9S12 software could need to implement a digital scale. Assume the range of a position measurement system is 0 to 3 m, and the system uses the MC9S12’s ADC to perform the measurement. The 10-bit ADC analog input range is 0 to +5 V, and the ADC digital output varies from 0 to 1023. Let \( x \) be the distance to be measured in metres, \( V_{\text{in}} \) be the analog voltage in volts and \( N \) be the 10-bit digital ADC output. Then the equations that relate the variables are:

\[
V_{\text{in}} = \frac{5 \cdot N}{1023} \quad \text{and} \quad x = 3 \cdot \frac{V_{\text{in}}}{5 \text{ V}}
\]

Thus:

\[
x = 3 \cdot \frac{N}{1023} = 0.0029325513 \cdot N \quad \text{where } x \text{ is in m}
\]

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.003 m. In other words, the distance must increase or decrease by 0.003 m for the digital output of the ADC to change by at least one number. It would be inappropriate to save the distance as an integer, because the only integers in this range are 0, 1, 2 and 3. Since the MC9S12 does not support floating-point, the distance data will be saved in fixed-point format. Decimal fixed-point is chosen because the distance data for this distance-meter will be displayed for a human to read. A fixed-point resolution of 0.001 m could be chosen, because it matches the resolution determined by the hardware.

The table below shows the performance of the system with the resolution set to 0.001 m. The table shows us that we need to store the fixed-point number in a signed or an unsigned 16-bit variable.

<table>
<thead>
<tr>
<th>( x ) (m)</th>
<th>( V_{\text{in}} ) (V) analog input</th>
<th>( N ) ADC input</th>
<th>( I ) internal representation</th>
<th>Approximation ( (44 \cdot N + 7) / 15 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.003</td>
<td>0.005</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0.600</td>
<td>1.000</td>
<td>205</td>
<td>600</td>
<td>601</td>
</tr>
<tr>
<td>1.500</td>
<td>2.500</td>
<td>512</td>
<td>1500</td>
<td>1502</td>
</tr>
<tr>
<td>3.000</td>
<td>5.000</td>
<td>1023</td>
<td>3000</td>
<td>3001</td>
</tr>
</tbody>
</table>
It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is overflow, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug:

\[ I = \frac{3000 \times N}{1023}; \]

because when \( N \) is greater than 21, \( 3000 \times N \) exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an approximate calculation can be performed without overflow

\[ I = \frac{44 \times N}{15}; \]

You can add one-half of the divisor to the dividend to implement rounding. In this case:

\[ I = \frac{44 \times N + 7}{15}; \]

The addition of “7” has the effect of rounding to the closest integer. The value 7 is selected because it is about one half of the divisor.

For example, when \( N = 4 \), the calculation \( \frac{44 \times 4}{15} = 11 \), whereas the \( \frac{(44 \times 4 + 7)}{15} \) calculation yields the better answer of 12.

No overflow occurs with this equation using unsigned 16-bit maths, because the maximum value of \( 44 \times N \) is 45012. If you can not rework the problem to eliminate overflow, the best solution is to use promotion. Promotion is the process of performing the operation in a higher precision. For example, in C we cast the input as unsigned long, and cast the result as unsigned short:

\[ I = \text{(unsigned short)}\left(\frac{3000 \times \text{(unsigned long)}N}{1023}\right); \]
Again, you can add one-half of the divisor to the dividend to implement rounding. In this case:

\[ I = (\text{unsigned short})\left(\frac{3000 \times (\text{unsigned long}N + 512)}{1023}\right) \]

The above equation will run slowly on a MC9S12 because there are no instructions to implement 32-bit by 32-bit arithmetic. When speed is important we can implement the calculation in assembly:

```assembly
ldd N
ldx #3000
emul ;32-bit Y:D is 3000*N
ldx #1023
ediv ;16-bit Y is (3000*N)/1023
sty I
```

The other error is called *drop out*. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. It is very important to divide last when performing multiple integer calculations. If you divided first:

\[ I = 44 \times (N / 15); \]

then the values of \( I \) would be only 0, 44, 88, … or 2992.

The display algorithm for the unsigned decimal fixed-point number with 0.001 resolution is simple:

1) display \((I / 1000)\) as a single digit value

2) display a decimal point

3) display \((I \% 1000)\) as a three-digit value

4) display the units “m”
Fixed-Point Operations Using a Universal 16Q8 Notation

Finding the optimum choice of Q notation for a fixed-point variable requires knowing what range of values it will have during execution. When range and resolution requirements are modest, however, a simple approach is to use 16 bits for all fixed-point numbers, with 8 bits in both the whole and the fractional parts, i.e. a 16Q8 notation:

With all operands using the same notation, addition and subtraction no longer require pre-alignment of operands. Multiplication and division, however, still require some adjustment or else the result will not have the same notation as the operands. Remember that when you multiply two fixed-point operands together, their Q notations add:

\[ 16Q8 \times 16Q8 = 32Q16 \] (10.5)
What we need is a product in 16Q8 format. In other words, the integer product needs to be right-shifted by 8 bits. Multiplying the two 16-bit integers produces a 32-bit product, with an implied binary point in the middle. Right-shifting this product by 8 bits and then putting the result back into a 16-bit location means that we are discarding 8 bits from each end of the integer product:

![Figure 10.7](image)

Discarding the least significant 8 bits simply causes some loss of precision; discarding the most significant 8 bits requires imposing a maximum magnitude restriction on the operands to avoid overflow.

When you divide one 16Q8 fixed-point operand by another, we require the result to be a 16Q8 number. We therefore need a 32Q16 dividend, since:

\[
32Q16 \div 16Q8 = 16Q8
\]

We create a 32Q16 dividend by sign extending the original 16Q8 dividend, and then left-shifting by 8 bits. The division is then done with a 32-bit dividend and a 16-bit divisor, to give a 16-bit quotient:

![Figure 10.8](image)
Example

We can use fixed-point algorithms to perform complex operations using the integer functions of our MC9S12. For example, consider the following digital filter calculation:

\[ y = x - 0.0532672 \times x_1 + x_2 + 0.0506038 \times y_1 - 0.9025 \times y_2; \]

In this case, the variables \( y, y_1, y_2, x, x_1, \) and \( x_2 \) are all 16Q8 fixed-point integers, and we need to express the constants in 16Q8 fixed-point format. The value \(-0.0532672\) will be approximated by \(-0.0532672 \times 256 \approx 14\). The value \(0.0506038\) will be approximated by \(0.0506038 \times 256 \approx 13\). Lastly, the value \(-0.9025\) will be approximated by \(-0.9025 \times 256 \approx -231\). The fixed-point implementation of this digital filter is:

```
INT32 t1, t2, t3, t4;
t1 = -14L * (INT32)x1;
t2 = 13L * (INT32)y1;
t3 = -231L * (INT32)y2;
t4 = t1 + t2 - t3;
y = x + x2 + (INT16)(t4 >> 8);
```

Note that since we are using C types, we need to allocate space for a 32-bit product, and thus the constants are expressed with \( L \) and the 16-bit integer variables are promoted and sign-extended to 32-bits using a typecast. If we did not do this, then the multiplication of two 16-bit quantities may overflow the 16-bit storage space.

The approximations of the constants using 16Q8 notation may be unsuitable if they do not give us enough resolution. In that case, we have to sacrifice speed and use a different non-power-of-2 base or increase the resolution of the Q notation numbers.
**Square Root Algorithm for a Fixed-Point Processor**

The evaluation of the square root of a number using integer arithmetic is a common operation in many embedded systems. For example, in the calculation of RMS quantities, such as voltage and current, a square root is involved. Any time a complex number is used (such as in an FFT), it is convenient to know its magnitude, which involves Pythagoras’ Theorem and a square root operation.

To evaluate the square root of a number, we use Newton’s method to solve the equation:

\[ f(x) = R - x^2 = 0 \quad (10.7) \]

where \( R \) is the number whose square root we wish to evaluate. According to a first-order Taylor series approximation of any function, we have:

\[ f'(x + h) \approx f'(x) + hf''(x) \quad (10.8) \]

If we have an estimate of the square root, \( x_* \), then we can use the above formula to determine an \( h \) to add to \( x_* \), which will hopefully be a better estimate of the square root:

\[
\begin{align*}
  f'(x_* + h) &= 0 \\
  f'(x_*) + hf''(x_*) &= 0 \\
  h &= -\frac{f'(x_*)}{f''(x_*)} \\
  h &= \frac{R - x_*^2}{2x_*} \quad (10.9)
\end{align*}
\]
This process is then repeated in an iterative manner until a desired accuracy is reached:

\[
x_\ast = x_\ast + h
\]

\[
\lim_{n \to \infty} x_\ast = x
\]  

(10.10)

Applying the above analysis to Eq. (10.7) gives a formula for the new estimate of the square root as:

\[
x_\ast = x_\ast - \frac{f'(x_\ast)}{f''(x_\ast)}
\]

\[
= x_\ast - \frac{R - x_\ast^2}{-2x_\ast}
\]

\[
= x_\ast + \frac{R}{2x_\ast} - \frac{x_\ast}{2}
\]

\[
= \frac{x_\ast + \frac{R}{2x_\ast} \left( \frac{R}{x_\ast} + x_\ast \right)}{2}
\]  

(10.11)

This is easily performed in an integer processor and involves only one division, one addition and a shift, which is very efficient.

When calculating an RMS value, we can calculate Eq. (10.11) once every sample time, and use the previous RMS value as the initial estimate. We don’t need to iterate more than once since the previous RMS value will always be a good estimate of the current RMS value.

C maths libraries provide square root routines, but when we understand their operation, we can optimise our code for performance.
Example

The following C function calculates the approximate magnitude of a complex number.

```c
// Number of iterations to perform for square-root algorithm
const UINT8 NB_ITERATIONS = 5;

UINT16 Magnitude(INT16 real, INT16 imag)
{
    UINT32 magSquared;
    UINT16 mag;
    UINT8 i;

    magSquared = (UINT32)((INT32)real * (INT32)real
                           + (INT32)imag * (INT32)imag);

    // Initial guess = magSquared / 2
    mag = (UINT16)(magSquared / 2);

    // Estimate magnitude using Newton's method
    for (i = 0; i < NB_ITERATIONS; i++)
        mag = (UINT16)((magSquared / mag + mag) / 2);

    return (mag);
}
```

The function above will return an approximate result since the number of iterations is fixed. This may be acceptable in certain applications – otherwise the error between the square of the current root estimate and the original number to be squared can be used to terminate the iterations.

The function also contains two bugs:

1. The initial estimate of the magnitude may exceed the range of a UINT16.
2. Division by zero is not tested for or handled.

Obviously a more robust function would need to handle these sources of potential error.
Lecture 11 – Real-Time Operating Systems


Introduction

A real-time operating system (RTOS) for an embedded system simplifies the design of real-time software by allowing the application to be divided into multiple threads managed by the RTOS. The kernel of an embedded RTOS needs to support multithreading, preemption, and thread priority. The RTOS will also provide services to threads for communication, synchronization and coordination. A RTOS is to be used for a “hard” real-time system – i.e. threads have to be performed not only correctly but also in a timely fashion.

Operating systems for larger computers (such as the PC) are non-real-time operating systems and usually provide a much larger range of application services, such as memory management and file management which normally do not apply to embedded systems.
11.2

Real-Time Kernel Concepts

The following sections describe real-time kernel concepts.

Threads

A thread is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into threads which are responsible for a portion of the problem. Each thread is assigned a priority, its own set of CPU registers and its own stack area.

Each thread is typically an infinite loop that can be in one of four states: READY, RUNNING, WAITING or INTERRUPTED.

A thread is READY when it can execute but its priority is less than the current running thread. A thread is RUNNING when it has control of the CPU. A thread is WAITING when the thread suspends itself until a certain amount of time has elapsed, or when it requires the occurrence of an event: waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur etc. Finally, a thread is INTERRUPTED when an interrupt occurred and the CPU is in the process of servicing the interrupt.

Figure 11.1 – Thread states
11.3

Context Switch

When the multithreading kernel decides to run a different thread, it simply saves the current thread’s context (CPU registers) in the current thread’s context storage area (the thread control block, or TCB). Once this operation is performed, the new thread’s context is restored from its TCB and the CPU resumes execution of the new thread’s code. This process is called a context switch. Context switching adds overhead to the application.

Kernel

The kernel is the part of an OS that is responsible for the management of threads (i.e., managing the CPU’s time) and for communication between threads. The fundamental service provided by the kernel is context switching.

Scheduler

The scheduler is the part of the kernel responsible for determining which thread will run next. Most real-time kernels are priority based. Each thread is assigned a priority based on its importance. Establishing the priority for each thread is application specific. In a priority-based kernel, control of the CPU will always be given to the highest priority thread ready to run. In a preemptive kernel, when a thread makes a higher priority thread ready to run, the current thread is preempted (suspended) and the higher priority thread is immediately given control of the CPU. If an interrupt service routine (ISR) makes a higher priority thread ready, then when the ISR is completed the interrupted thread is suspended and the new higher priority thread is resumed.

![Figure 11.2 – Preemptive kernel](image)

Embedded Software 2014
With a preemptive kernel, execution of the highest priority thread is deterministic; you can determine when the highest priority thread will get control of the CPU.

Application code using a preemptive kernel should not use non-reentrant functions, unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low- and a high-priority thread can use a common function. Corruption of data may occur if the higher priority thread preempts a lower priority thread that is using the function.

To summarize, a preemptive kernel always executes the highest priority thread that is ready to run. An interrupt preempts a thread. Upon completion of an ISR, the kernel resumes execution to the highest priority thread ready to run (not the interrupted thread). Thread-level response is optimum and deterministic.
Reentrancy

A reentrant function can be used by more than one thread without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used. An example of a reentrant function is shown below:

```c
char * strcpy(char *dst, const char *src)
{
    char *ptr = dst;
    while (*dst++ = *src++);
    return ptr;
}
```

Since copies of the arguments to `strcpy()` are placed on the thread's stack, and the local variable is created on the thread’s stack, `strcpy()` can be invoked by multiple threads without fear that the threads will corrupt each other's pointers.

An example of a non-reentrant function is shown below:

```c
int Temp;

void swap(int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

`swap()` is a simple function that swaps the contents of its two arguments. Since `Temp` is a global variable, if the `swap()` function gets preempted after the first line by a higher priority thread which also uses the `swap()` function, then when the low priority thread resumes it will use the `Temp` value that was used by the high priority thread.

You can make `swap()` reentrant with one of the following techniques:

- Declare `Temp` local to `swap()`.
- Disable interrupts before the operation and enable them afterwards.
- Use a semaphore.
Thread Priority

A priority is assigned to each thread. The more important the thread, the higher the priority given to it.

Static Priorities

Thread priorities are said to be static when the priority of each thread does not change during the application's execution. Each thread is thus given a fixed priority at compile time. All the threads and their timing constraints are known at compile time in a system where priorities are static.

Dynamic Priorities

Thread priorities are said to be dynamic if the priority of threads can be changed during the application's execution; each thread can change its priority at run time. This is a desirable feature to have in a real-time kernel to avoid priority inversions.

Priority Inversions

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel. Priority inversion is any situation in which a low priority thread holds a resource while a higher priority thread is ready to use it. In this situation the low priority thread prevents the high priority thread from executing until it releases the resource.

To avoid priority inversion a multithreading kernel should change the priority of a thread automatically to help prevent priority inversions. This is called priority inheritance.
Mutual Exclusion

The easiest way for threads to communicate with each other is through shared data structures. This is especially easy when all threads exist in a single address space and can reference global variables, pointers, buffers, linked lists, FIFOs, etc. Although sharing data simplifies the exchange of information, you must ensure that each thread has exclusive access to the data to avoid contention and data corruption. The most common methods of obtaining exclusive access to shared resources are:

- disabling interrupts,
- performing test-and-set operations,
- disabling scheduling, and
- using semaphores.

Disabling and Enabling Interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts, as shown in the pseudocode:

```c
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

Kernels use this technique to access internal variables and data structures. In fact, kernels usually provide two functions that allow you to disable and then enable interrupts from your C code: `OS_EnterCritical()` and `OS_ExitCritical()`, respectively. You need to use these functions in tandem, as shown below:

```c
void Function(void)
{
    OS_EnterCritical();
    /* You can access shared data in here */
    OS_ExitCritical();
}
```

You must be careful, however, not to disable interrupts for too long because this affects the response of your system to interrupts. This is known as interrupt
latency. You should consider this method when you are changing or copying a few variables. Also, this is the only way that a thread can share variables or data structures with an ISR. In all cases, you should keep interrupts disabled for as little time as possible.

If you use a kernel, you are basically allowed to disable interrupts for as much time as the kernel does without affecting interrupt latency. Obviously, you need to know how long the kernel will disable interrupts.

Semaphores

The semaphore was invented by Edgser Dijkstra in the mid-1960s. It is a protocol mechanism offered by most multithreading kernels. Semaphores are used to:

- control access to a shared resource (mutual exclusion),
- signal the occurrence of an event, and
- allow two threads to synchronize their activities.

A semaphore is a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting thread is suspended until the semaphore is released by its current owner. In other words, the requesting thread says: "Give me the key. If someone else is using it, I am willing to wait for it!" There are two types of semaphores: binary semaphores and counting semaphores. As its name implies, a binary semaphore can only take two values: 0 or 1. A counting semaphore allows values between 0 and 255, 65535, or 4294967295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. The actual size depends on the kernel used. Along with the semaphore's value, the kernel also needs to keep track of threads waiting for the semaphore's availability.

Generally, only three operations can be performed on a semaphore: Init(), Wait(), and Signal(). The initial value of the semaphore must be
provided when the semaphore is initialized. The waiting list of threads is always initially empty.

A thread desiring the semaphore will perform a `Wait()` operation. If the semaphore is available (the semaphore value is greater than 0), the semaphore value is decremented and the thread continues execution. If the semaphore's value is 0, the thread performing a `Wait()` on the semaphore is placed in a waiting list. Most kernels allow you to specify a timeout; if the semaphore is not available within a certain amount of time, the requesting thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller.

A thread releases a semaphore by performing a `Signal()` operation. If no thread is waiting for the semaphore, the semaphore value is simply incremented. If any thread is waiting for the semaphore, however, one of the threads is made ready to run and the semaphore value is not incremented; the key is given to one of the threads waiting for it. Depending on the kernel, the thread that receives the semaphore is either:

- the highest priority thread waiting for the semaphore, or
- the first thread that requested the semaphore (First In First Out).

Some kernels have an option that allows you to choose either method when the semaphore is initialized. For the first option, if the readied thread has a higher priority than the current thread (the thread releasing the semaphore), a context switch occurs (with a preemptive kernel) and the higher priority thread resumes execution; the current thread is suspended until it again becomes the highest priority thread ready to run.

Listing 11.1 shows how you can share data using a semaphore. Any thread needing access to the same shared data calls `OS_SemaphoreWait()`, and when the thread is done with the data, the thread calls `OS_SemaphoreSignal()`. Both of these functions are described later. You should note that a semaphore is an object that needs to be initialized before it is
used; for mutual exclusion, a semaphore is initialized to a value of 1. Using a semaphore to access shared data doesn't affect interrupt latency. If an ISR or the current thread makes a higher priority thread ready to run while accessing shared data, the higher priority thread executes immediately.

```c
OS_EVENT *SharedDataSemaphore;

void Function(void)
{
    UINT8 error;

    error = OS_SemaphoreWait(SharedDataSemaphore, 0);
    // You can access shared data in here
    // (interrupts are recognized)
    error = OS_SemaphoreSignal(SharedDataSemaphore);
}
```

Listing 11.1 – Accessing shared data by obtaining a semaphore

Semaphores are especially useful when threads share I/O devices. Imagine what would happen if two threads were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each thread. For instance, the printout from Thread 1 printing "I am Thread 1!" and Thread 2 printing "I am Thread 2!" could result in:

"I I a amm T Threahread dl !2!"

In this case, use a semaphore and initialize it to 1 (i.e., a binary semaphore). The rule is simple: to access the printer each thread first must obtain the resource's semaphore.
Figure 11.3 shows threads competing for a semaphore to gain exclusive access to the printer. Note that the semaphore is represented symbolically by a key, indicating that each thread must obtain this key to use the printer.

The above example implies that each thread must know about the existence of the semaphore in order to access the resource. There are situations when it is better to encapsulate the semaphore. Each thread would thus not know that it is actually acquiring a semaphore when accessing the resource. For example, the SCI port may be used by multiple threads to send commands and receive responses from a PC:
The function `Packet_Put()` is called with two arguments: the packet and a timeout in case the device doesn't respond within a certain amount of time. The pseudocode for this function is shown in Listing 11.2.

```c
UINT8 Packet_Put(TPacket *packet, UINT16 timeout)
{
    Acquire serial port's semaphore;
    Send packet to device;
    Wait for response (with timeout);
    Release semaphore;
    if (timed out)
        return (error code);
    else
        return (no error);
}
```

Listing 11.2 – Encapsulating a semaphore

Each thread that needs to send a packet to the serial port has to call this function. The semaphore is assumed to be initialized to 1 (i.e., available) by the communication driver initialization routine. The first thread that calls `Packet_Put()` acquires the semaphore, proceeds to send the packet, and waits for a response. If another thread attempts to send a command while the port is busy, this second thread is suspended until the semaphore is released. The second thread appears simply to have made a call to a normal function that will not return until the function has performed its duty. When the semaphore is released by the first thread, the second thread acquires the semaphore and is allowed to use the SCI port.
A counting semaphore is used when a resource can be used by more than one thread at the same time. For example, a counting semaphore is used in the management of a buffer pool as shown in Figure 11.5.

Assume that the buffer pool initially contains 10 buffers. A thread would obtain a buffer from the buffer manager by calling `Buffer_Request()`. When the buffer is no longer needed, the thread would return the buffer to the buffer manager by calling `Buffer_Release()`. The pseudocode for these functions is shown in Listing 11.3.

```c
BUF *Buffer_Request(void)
{
    BUF *ptr;

    Acquire a semaphore;
    Disable interrupts;
    ptr = BufFreeList;
    BufFreeList = ptr->next;
    Enable interrupts;
    return (ptr);
}
```
void Buffer_Release(BUF *ptr)
{
    Disable interrupts;
    ptr->next = BufFreeList;
    BufFreeList = ptr;
    Enable interrupts;
    Release semaphore;
}

Listing 11.3 – Buffer management using a semaphore

The buffer manager will satisfy the first 10 buffer requests because there are 10 keys. When all semaphores are used, a thread requesting a buffer is suspended until a semaphore becomes available. Interrupts are disabled to gain exclusive access to the linked list (this operation is very quick). When a thread is finished with the buffer it acquired, it calls Buffer_Release() to return the buffer to the buffer manager; the buffer is inserted into the linked list before the semaphore is released. By encapsulating the interface to the buffer manager in Buffer_Request() and Buffer_Release(), the caller doesn't need to be concerned with the actual implementation details.

Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in most situations. The overhead involved in acquiring and releasing the semaphore can consume valuable time. You can do the job just as efficiently by disabling and enabling interrupts. Suppose that two threads are sharing a 32-bit integer variable. The first thread increments the variable while the other thread clears it. If you consider how long a processor takes to perform either operation, you will realize that you do not need a semaphore to gain exclusive access to the variable. Each thread simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete. A semaphore should be used, however, if the variable is a floating-point variable and the microprocessor doesn't support floating point in hardware. In this case, the processing time involved in processing the floating-point variable could have affected interrupt latency if you had disabled interrupts.
Deadlock (or Deadly Embrace)

A deadlock, also called a deadly embrace, is a situation in which two threads are each unknowingly waiting for resources held by the other. Assume thread T1 has exclusive access to resource R1 and thread T2 has exclusive access to resource R2. If T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither thread can continue. They are deadlocked. The simplest way to avoid a deadlock is for threads to:

- acquire all resources before proceeding,
- acquire the resources in the same order, and
- release the resources in the reverse order

Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken. If the semaphore is not available within a certain amount of time, the thread requesting the resource resumes execution. Some form of error code must be returned to the thread to notify it that a timeout occurred. A return error code prevents the thread from thinking it has obtained the resource. Deadlocks generally occur in large multithreading systems, not in embedded systems.
Synchronization

A thread can be synchronized with an ISR (or another thread when no data is being exchanged) by using a semaphore as shown in Figure 11.6.

![Figure 11.6 – Synchronizing threads and ISRs](image)

Note that, in this case, the semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key). When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for this type of synchronization is called a *unilateral rendezvous*. A thread initiates an I/O operation and waits for the semaphore. When the I/O operation is complete, an ISR (or another thread) signals the semaphore and the thread is resumed.

If the kernel supports counting semaphores, the semaphore would accumulate events that have not yet been processed. Note that more than one thread can be waiting for an event to occur. In this case, the kernel could signal the occurrence of the event either to:

- the highest priority thread waiting for the event to occur or
- the first thread waiting for the event.

Depending on the application, more than one ISR or thread could signal the occurrence of the event.
Two threads can synchronize their activities by using two semaphores, as shown in Figure 11.7. This is called a *bilateral rendezvous*. A bilateral rendezvous is similar to a unilateral rendezvous, except both threads must synchronize with one another before proceeding.

![Figure 11.7 – Threads synchronizing their activities](image)

For example, two threads are executing as shown in Listing 11.4. When the first thread reaches a certain point, it signals the second thread (1) then waits for a return signal (2). Similarly, when the second thread reaches a certain point, it signals the first thread (3) and waits for a return signal (4). At this point, both threads are synchronized with each other. A bilateral rendezvous cannot be performed between a thread and an ISR because an ISR cannot wait on a semaphore.

```c
void Thread1(void)
{
    for (;;)
    {
        Perform operation 1;
        Signal thread #2;               (1)
        Wait for signal from thread #2; (2)
        Continue operation 1;
    }
}

void Thread2(void)
{
    for (;;)
    {
        Perform operation 2;
        Signal thread #1;               (3)
        Wait for signal from thread #1; (4)
        Continue operation 2;
    }
}
```

*Listing 11.4 – Bilateral rendezvous*
Interthread Communication

It is sometimes necessary for a thread or an ISR to communicate information to another thread. This information transfer is called interthread communication. Information may be communicated between threads in two ways: through global data or by sending messages.

When using global variables, each thread or ISR must ensure that it has exclusive access to the variables. If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts. If two threads are sharing data, each can gain exclusive access to the variables either by disabling and enabling interrupts or with the use of a semaphore (as we have seen). Note that a thread can only communicate information to an ISR by using global variables. A thread is not aware when a global variable is changed by an ISR, unless the ISR signals the thread by using a semaphore or unless the thread polls the contents of the variable periodically. To correct this situation, you should consider using either a message mailbox or a message queue.
Message Mailboxes

Messages can be sent to a thread through kernel services. A Message Mailbox, also called a message exchange, is typically a pointer-size variable. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree on what the pointer is actually pointing to.

A waiting list is associated with each mailbox in case more than one thread wants to receive messages through the mailbox. A thread desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to it. When a message is deposited into the mailbox, either the highest priority thread waiting for the message is given the message (priority based) or the first thread to request a message is given the message (First-In-First-Out, or FIFO). Figure 11.8 shows a thread depositing a message into a mailbox. Note that the mailbox is represented by an I-beam and the timeout is represented by an hourglass. The number next to the hourglass represents the number of clock ticks the thread will wait for a message to arrive.

![Figure 11.8 – Message mailbox](image-url)
Kernels typically provide the following mailbox services:

- Initialize the contents of a mailbox. The mailbox initially may or may not contain a message.

- Deposit a message into the mailbox (POST).

- Wait for a message to be deposited into the mailbox (WAIT).

- Get a message from a mailbox if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox. A return code is used to notify the caller about the outcome of the call.

Message mailboxes can also simulate binary semaphores. A message in the mailbox indicates that the resource is available, and an empty mailbox indicates that the resource is already in use by another thread.

**Message Queues**

A message queue is used to send one or more messages to a thread. A message queue is basically an array of mailboxes. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into a message queue. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree as to what the pointer is actually pointing to. Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).

As with the mailbox, a waiting list is associated with each message queue, in case more than one thread is to receive messages through the queue. A thread desiring a message from an empty queue is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating a timeout has occurred) is returned to it. When a message is deposited into the queue, either the highest priority thread or the first thread to
wait for the message is given the message. Figure 11.9 shows an ISR (Interrupt Service Routine) depositing a message into a queue. Note that the queue is represented graphically by a double I-beam. The "10" indicates the number of messages that can accumulate in the queue. A "0" next to the hourglass indicates that the thread will wait forever for a message to arrive.

Kernels typically provide the message queue services listed below.

- Initialize the queue. The queue is always assumed to be empty after initialization.
- Deposit a message into the queue (POST).
- Wait for a message to be deposited into the queue (WAIT).
- Get a message from a queue if one is present, but do not suspend the caller if the queue is empty (ACCEPT). If the queue contains a message, the message is extracted from the queue. A return code is used to notify the caller about the outcome of the call.
Interrupts

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves all of its context (i.e., registers) and jumps to a special subroutine called an Interrupt Service Routine, or ISR. The ISR processes the event, and upon completion of the ISR, the program returns to:

- the background for a foreground / background system,
- the interrupted thread for a non-preemptive kernel, or
- the highest priority thread ready to run for a preemptive kernel.

Interrupts allow a microprocessor to process events when they occur. This prevents the microprocessor from continuously polling an event to see if it has occurred. Microprocessors allow interrupts to be ignored and recognized through the use of two special instructions: disable interrupts and enable interrupts, respectively. In a real-time environment, interrupts should be disabled as little as possible. Disabling interrupts affects interrupt latency and may cause interrupts to be missed. Processors generally allow interrupts to be nested. This means that while servicing an interrupt, the processor will recognize and service other (more important) interrupts, as shown in Figure 11.10.

![Figure 11.10 – Interrupt nesting](image)
**Interrupt Latency**

Probably the most important specification of a real-time kernel is the amount of time interrupts are disabled. All real-time systems disable interrupts to manipulate critical sections of code and reenable interrupts when the critical section has executed. The longer interrupts are disabled, the higher the interrupt latency. Interrupt latency is given by Eq. (11.1).

\[
\text{Interrupt latency} = \text{Maximum amount of time interrupts are disabled} + \text{Time to start executing the first instruction in the ISR}
\]  \hspace{1cm} (11.1)

**Interrupt Response**

Interrupt response is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt. The interrupt response time accounts for all the overhead involved in handling an interrupt.

For a foreground / background system, the user ISR code is executed immediately. The response time is given by Eq. (11.2).

\[
\text{Interrupt response time} = \text{Interrupt latency}
\]  \hspace{1cm} (11.2)

For a preemptive kernel, a special function provided by the kernel needs to be called. This function notifies the kernel that an ISR is in progress and allows the kernel to keep track of interrupt nesting. This function is called `OS_ISREnter()`. The response time to an interrupt for a preemptive kernel is given by Eq. (11.3).

\[
\text{Interrupt response time} = \text{Interrupt latency} + \text{Execution time of the kernel ISR entry function}
\]  \hspace{1cm} (11.3)
A system's worst case interrupt response time is its only response time. Your system may respond to interrupts in 50ms 99 percent of the time, but if it responds to interrupts in 250ms the other 1 percent, you must assume a 250ms interrupt response time.

**Interrupt Recovery**

Interrupt recovery is defined as the time required for the processor to return to the interrupted code. Interrupt recovery in a foreground / background system simply involves restoring the processor's context and returning to the interrupted thread. Interrupt recovery is given by Eq. (11.4).

\[
\text{Interrupt recovery time} = \text{Time to execute the return from interrupt instruction} \quad (11.4)
\]

For a preemptive kernel, interrupt recovery is more complex. Typically, a function provided by the kernel is called at the end of the ISR. This function is called `OS_ISRExit()` and allows the kernel to determine if all interrupts have nested. If they have nested (i.e., a return from interrupt would return to thread-level code), the kernel determines if a higher priority thread has been made ready to run as a result of the ISR. If a higher priority thread is ready to run as a result of the ISR, this thread is resumed. Note that, in this case, the interrupted thread will resume only when it again becomes the highest priority thread ready to run. For a preemptive kernel, interrupt recovery is given by Eq. (11.5).

\[
\text{Interrupt recovery time} = \text{Time to determine if a higher priority thread is ready} \\
+ \text{Time to restore the CPU's context of the highest priority thread} \\
+ \text{Time to execute the return from interrupt instruction} \quad (11.5)
\]
Interrupt Latency, Response, and Recovery

Figure 11.11 and Figure 11.12 show the interrupt latency, response, and recovery for a foreground / background system and a preemptive kernel, respectively.

**Figure 11.11 – Interrupt latency, response, and recovery (foreground / background)**
You should note that for a preemptive kernel, the exit function either decides to return to the interrupted thread (A) or to a higher priority thread that the ISR has made ready to run (B). In the latter case, the execution time is slightly longer because the kernel has to perform a context switch.

![Figure 11.12 – Interrupt latency, response, and recovery (preemptive kernel)](image)

**ISR Processing Time**

Although ISRs should be as short as possible, there are no absolute limits on the amount of time for an ISR. One cannot say that an ISR must always be less than 100 ms, 500 ms, or 1 ms. If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be. In most cases, however, the ISR should recognize the interrupt, obtain data or a status from the interrupting device, and signal a thread to perform the actual processing. You should also consider whether the overhead involved in signalling a thread is more than the processing of the interrupt. Signalling a thread from an ISR (i.e., through a semaphore, a mailbox, or a queue) requires some processing time. If processing your interrupt requires less than the time required to signal a thread, you should consider processing the interrupt in the ISR itself and possibly enabling interrupts to allow higher priority interrupts to be recognized and serviced.
Clock Tick

A clock tick is a special interrupt that occurs periodically. This interrupt can be viewed as the system's heartbeat. The time between interrupts is application specific and is generally between 1 and 200 ms. The clock tick interrupt allows a kernel to delay threads for an integral number of clock ticks and to provide timeouts when threads are waiting for events to occur. The faster the tick rate, the higher the overhead imposed on the system.

All kernels allow threads to be delayed for a certain number of clock ticks. The resolution of delayed threads is one clock tick; however, this does not mean that its accuracy is one clock tick.

Figure 11.13 through Figure 11.15 are timing diagrams showing a thread delaying itself for one clock tick. The shaded areas indicate the execution time for each operation being performed. Note that the time for each operation varies to reflect typical processing, which would include loops and conditional statements (i.e., if/else, switch, and ?:). The processing time of the Tick ISR has been exaggerated to show that it too is subject to varying execution times.

![Diagram of Clock Tick and Thread Delay](image)

Figure 11.13 – Delaying a thread for one tick (Case 1)

Case 1 (Figure 11.13) shows a situation where higher priority threads and ISRs execute prior to the thread, which needs to delay for one tick. The thread
attempts to delay for 20ms but because of its priority, it actually executes at varying intervals. This causes the execution of the thread to jitter.

Figure 11.14 – Delaying a thread for one tick (Case 2)

Case 2 (Figure 11.14) shows a situation where the execution times of all higher priority threads and ISRs are slightly less than one tick. If the thread delays itself just before a clock tick, the thread will execute again almost immediately! Because of this, if you need to delay a thread at least one clock tick, you must specify one extra tick. In other words, if you need to delay a thread for at least five ticks, you must specify six ticks!

Figure 11.15 – Delaying a thread for one tick (Case 3)

Case 3 (Figure 11.15) shows a situation in which the execution times of all higher priority threads and ISRs extend beyond one clock tick. In this case, the thread that tries to delay for one tick actually executes two ticks later and
misses its deadline. This might be acceptable in some applications, but in most cases it isn't.

These situations exist with all real-time kernels. They are related to CPU processing load and possibly incorrect system design. Here are some possible solutions to these problems:

- Increase the clock rate of your microprocessor.
- Increase the time between tick interrupts.
- Rearrange thread priorities.
- Avoid using floating-point maths (if you must, use single precision).
- Get a compiler that performs better code optimization.
- Write time-critical code in assembly language.
- If possible, upgrade to a faster microprocessor in the same family; that is, MC9S12 to MC9SX12, etc.

Regardless of what you do, jitter will always occur.

**Memory Requirements**

If you are designing a foreground / background system, the amount of memory required depends solely on your application code. With a multithreading kernel, things are quite different. To begin with, a kernel requires extra code space (Flash). The size of the kernel depends on many factors. Depending on the features provided by the kernel, you can expect anywhere from 1 to 100 Kb. A minimal kernel for a 16-bit CPU that provides only scheduling, context switching, semaphore management, delays, and timeouts should require about 1 to 3 Kb of code space.

Because each thread runs independently of the others, it must be provided with its own stack area (RAM). As a designer, you must determine the stack requirement of each thread as closely as possible (this is sometimes a difficult
The stack size must not only account for the thread requirements (local variables, function calls, etc.), it must also account for maximum interrupt nesting (saved registers, local storage in ISRs, etc.). Depending on the target processor and the kernel used, a separate stack can be used to handle all interrupt-level code. This is a desirable feature because the stack requirement for each thread can be substantially reduced. Another desirable feature is the ability to specify the stack size of each thread on an individual basis. Conversely, some kernels require that all thread stacks be the same size. All kernels require extra RAM to maintain internal variables, data structures, queues, etc. The total RAM required if the kernel does not support a separate interrupt stack is given by Eq. (11.6).

\[
\text{Total RAM requirements} = \text{Application code requirements} + \text{Data space (i.e., RAM) needed by the kernel} + \text{SUM(thread stacks + MAX(ISR nesting))} \quad (11.6)
\]

Unless you have large amounts of RAM to work with, you need to be careful how you use the stack space. To reduce the amount of RAM needed in an application, you must be careful how you use each thread's stack for:

- large arrays and structures declared locally to functions and ISRs,
- function (i.e., subroutine) nesting,
- interrupt nesting,
- library functions stack usage, and
- function calls with many arguments.

To summarize, a multithreading system requires more code space (Flash) and data space (RAM) than a foreground / background system. The amount of extra Flash depends only on the size of the kernel, and the amount of RAM depends on the number of threads in your system.

Embedded Software 2014
Advantages and Disadvantages of Real-Time Operating Systems

An RTOS allows real-time applications to be designed and expanded easily; functions can be added without requiring major changes to the software. The use of an RTOS simplifies the design process by splitting the application code into separate threads. With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible. An RTOS allows you to make better use of your resources by providing you with valuable services, such as semaphores, mailboxes, queues, time delays, timeouts, etc.

You should consider using a real-time kernel if your application can afford the extra requirements: extra cost of the kernel, more ROM/RAM, and 2 to 4 percent additional CPU overhead.
Software Style Guide


This document gives an overview of the software style to be used when programming in C for an embedded system.

Quality Programming

Software engineering is like other fields of engineering. Engineering is about implementing the solutions to important problems by creatively applying methods from sound bodies of scientific theory and by experimentation – and doing it for the benefit of members of society (with minimal impact in terms of economic, environmental, and societal cost). The fact that engineers need to be creative to find solutions to problems leads to many interesting and novel ideas – which generally advances the state-of-the-art. This creativity is the very reason that engineering is carried out by humans (at the present).

Creativity leads to one major problem: there are a very large number of ways to implement a solution to a problem. We need to be guided by theory, practice and past experience in seeking out whether a particular solution meets the specifications.

Large systems, especially software systems, tend to involve a level of complexity that is beyond the capability of one person. The only way to build and maintain large, complex systems is by following well-defined procedures during the engineering development cycle. One of those procedures is engineering design. There are many procedures we can use for software engineering design – block diagrams, data flow graphs, UML etc.

The most important phase of the engineering development cycle is implementation. In an embedded system this involves both hardware and software. The ultimate goal of an embedded system is to meet the stated objectives such as functionality, input/output relationships, stability and accuracy. Nevertheless it is appropriate to separately evaluate the individual
components of a system. Software quality is one key area that needs to be evaluated.

There are two categories of performance criteria with which we evaluate software. *Quantitative* criteria include static efficiency (e.g., memory requirements), dynamic efficiency (e.g., speed of execution), and accuracy of the results. *Qualitative* criteria centre around ease of understanding. If your software is easy to understand then it will be:

- Easy to debug (fix mistakes)
- Easy to verify (prove correctness)
- Easy to maintain (add features)

Since there is no “best way” to write software, this document simply outlines techniques, based on experience, that you should try to adopt when forming your own software style. In particular, the style of writing software presented leads to code that is: self-documenting, modular, and layered.

You can tell if you write good software if:

1) you can understand your own code one year later
2) others can make changes to your code.
Naming Convention

1. *Names should have meaning*

   If we observe a name out of the context of the program in which it exists, the meaning of the object should be obvious. The object TxFIFO is clearly the transmit first in first out circular buffer. The function LCD_OutString will output a string to the LCD display.

2. *Avoid ambiguities*

   Don't use variable names that are vague or have more than one meaning. For example, it is vague to use temp, because there are many possibilities for temporary data, in fact, it might even mean temperature. Don't use two names that look similar, but have different meanings.

3. *Give hints about the type*

   We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, dataPtr, timePtr, putPtr are pointers. Similarly, voltageBuf, timeBuf, pressureBuf, are data buffers. Other good phrases include Flag, Mode, U, L, Index, Nb, which refer to boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a number (counter) respectively.

4. *Use the same name to refer to the same type of object*

   For example, everywhere we need a local variable to store an ASCII character we could use the name letter. Another common example is to use the names i, j, k for indices into arrays. The names V1 and R1 might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just the fact that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.
5. Use a prefix to identify public objects

An underline character should separate the module name from the function name. As an exception to this rule, we can use the underline to delimit words in an all upper-case name (e.g., \#define MIN\_PRESSURE 10). Functions that can be accessed outside the scope of a module should begin with a prefix specifying the module to which it belongs. It is poor style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the file name containing the object. For example, if we see a function call, LCD\_OutString("Hello world"); we know the public function belongs to the LCD module, where the policies are defined in LCD.h and the implementation in LCD.c. Notice the similarity between this syntax (e.g., LCD\_Init()) and the corresponding syntax we would use if programming the module as a class in C++ (e.g., LCD.Init()). Using this convention, we can easily distinguish public and private objects. If the variable is public, because the name has an underline, then the first letter of the name after the underline should be capitalized (e.g., my\_Count is a public variable belonging to the module “my” and defined in the header file my.h).

6. Use upper- and lower-case to specify the scope of an object

We will define I/O ports and constants using upper-case letters. In other words, names with upper-case letters refer to objects with fixed values. TRUE, FALSE and NULL are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words will use an underline character to delimit the individual words, e.g., MAX\_VOLTAGE, UPPER\_BOUND, FIFO\_SIZE. Global objects will begin with a capital letter, but include some lower-case letters. Local variables will begin with a lower-case letter, and may or may not include upper-case letters. Since all functions are global, we can start function names with either an upper-case or lower-case letter. Using this convention, we can distinguish constants, globals and locals.
7. Use capitalization to delimit words

Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether it is local or global. Some programmers use the *underline* as a word-delimiter, but except for constants we will reserve *underline* to separate the module name from the variable name.

<table>
<thead>
<tr>
<th>type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>CR, SAFE_TO_RUN, PORTA, STACK_SIZE, START_OF_RAM</td>
</tr>
<tr>
<td>local variable</td>
<td>maxTemperature, lastCharTyped, errorCount</td>
</tr>
<tr>
<td>private global variable</td>
<td>MaxTemperature, LastCharTyped, ErrorCount</td>
</tr>
<tr>
<td>public global variable</td>
<td>DAC_MaxTemperature, Key_LastCharTyped, Network_ErrorCount, File_OpenFlag</td>
</tr>
<tr>
<td>private function</td>
<td>ClearTime, wrapPointer, InChar</td>
</tr>
<tr>
<td>public function</td>
<td>Timer_ClearTime, FIFO_Put, Key_InChar</td>
</tr>
</tbody>
</table>

Table S.1 – Examples of naming conventions
Code Style Structure (the .c file)

Maintaining a consistent style will help us locate and understand the different components of our software, as well as prevent us from forgetting to include a component (or worse, including it twice). The following regions should occur in this order in every code file (e.g., file.c).

1. Opening comments

The first line of every file should contain the file name. These opening comments will be duplicated in the corresponding header file (e.g., file.h) and are intended to be read by the client, the one who will use these programs. If major portions of this software are copied from copyrighted sources, then we must satisfy the copyright requirements of those sources. The rest of the opening comments should include:

- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module, and
- any copyright information.

2. Including .h files

Next, we will place the #include statements that add the necessary header files. Adding other code files, if necessary, will occur at the end of the file, but here at the top of the file we include just the header files. Normally the order doesn't matter, so we will list the include files in a hierarchical fashion starting with the lowest level and ending at the highest. If the order of these statements is important, then write a comment describing both what the proper order is and why the order is important. Putting them together at the top will help us draw a call-graph, which will show us how our modules are connected. In particular, if we consider each code file to be a separate module, then the list of #include statements specifies which other modules can be called from this module. Of course one header file is allowed to include other header files. Be careful to include only those files that are
absolutely necessary. Adding unnecessary include statements will make our system seem more complex than it actually is.

3. **extern references**

After including the header files, we can declare any external variables or functions. External references will be resolved by the linker, when various modules are linked together to create a single executable application. Placing them together at the top of the file will help us see how this software system fits together (i.e., is linked to) other software systems.

4. **#define statements**

After external references, we should place the `#define` macros and `#define` constants. Since these definitions are located in the code file (e.g., `file.c`), they will be private. This means they are available within this file only. If the client does not need to use or change the macro or constant, then it should be made private by placing it here in the code file. Conversely, if we wish to create a public constant or macro, then we place it in the header file for this module.

5. **struct, union, enum statements**

After the define statements, we should create the necessary data structures using `struct`, `union` and `enum`. Again, since these definitions are located in the code file (e.g., `file.c`), they will be private.

6. **Global variables and constants**

After the structure definitions, we should include the global variables and constants. If we specify the global as `static` then it will be private, and can only be accessed by programs in this file. If we do not specify the global as `static` then it will be public, and can be accessed by any program (that program defines it as `extern` and the linker will resolve the reference). We put all the globals together before any function definitions to symbolize the fact that any function in this file has access to these globals. If we have a
A permanent variable that is only accessed by one function, then it should be defined as a `static` local. The scope of a variable includes all the software in the system that can access it. In general, we wish to minimize the scope of our data.

<table>
<thead>
<tr>
<th>declaration</th>
<th>accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>short file_PublicGlobal;</code></td>
<td>by any function via <code>extern</code> declaration</td>
</tr>
<tr>
<td><code>static short PrivateGlobal;</code></td>
<td>in this file only</td>
</tr>
<tr>
<td><code>void function()</code></td>
<td>by this function only, but persistent (not on stack and initialised once only)</td>
</tr>
</tbody>
</table>

Table S.2 – Accessibility of variables

7. **Prototypes of private functions**

After the globals, we should add any necessary prototypes. Just like global variables, we can restrict access to private functions by defining them as `static`. Prototypes for the public functions will be included in the corresponding header file. In general, we will arrange the code implementations in a top-down fashion. Although not necessary, we will include the parameter names with the prototypes. Descriptive parameter names will help document the usage of the function.

8. **Implementations of the functions**

The heart of the implementation file will be, of course, the implementations. Again, private functions should be defined as `static`. The functions should be sequenced in a logical manner. The most typical sequence is top-down, meaning we begin with the highest level and finish with the lowest level. Another appropriate sequence mirrors the manner in which the functions will be used. For example, start with the initialization functions, followed by the operations, and end with the shutdown functions. For example:

```c
open();
input();
output();
close();
```
9. *Including .c files*

At the end of the file, we will place the `#include` statements that add the necessary code files. If our compiler supports projects, then it is a good idea to take advantage of this feature. The project simplifies the management of large software systems by providing organizational structure to the software system. If we use projects, then including code files will be unnecessary, and hence should be avoided. If our compiler does not support projects, or if we are writing software for multiple compilers, then including code files allows a large software project to be constructed simply by compiling the file with the `main()` program in it. Including header files at the top of the file allows this module to access public variables and functions of other modules. On the other hand, including code files at the end of the file prevents this module from accessing private variables and functions of the other modules.
Header Style Structure (the .h file)

Once again, maintaining a consistent style facilitates understanding and helps to avoid errors of omission. Definitions made in the header file will be public, i.e., accessible by all modules. As stated earlier, it is better to make global variables private rather than placing them in the header file. Similarly, we should avoid placing actual code in a header file.

There are two types of header files. The first type of header file has no corresponding code file. In other words, there is a file.h, but no file.c. In this type of header, we can list global constants and helper macros. Examples of global constants are I/O port addresses (e.g., mc9s12a512.h) and calibration coefficients. Debugging macros could be grouped together and placed in a debug.h file. We will not consider software in these types of header files as belonging to a particular module.

The second type of header file does have a corresponding code file. The two files, e.g., file.h, and file.c, form a software module. In this type of header, we define the prototypes for the public functions of the module. The file.h contains the policies (behaviour or what it does) and the file.c file contains the mechanisms (functions or how it works.) The following regions should occur in order in every header file (e.g., file.h).

1. **Opening comments**

   The first line of every file should contain the file name. These opening comments will be duplicated in the corresponding code file (e.g., file.c) and are intended to be read by the client, the one who will use these programs. We should repeat copyright information as appropriate. The rest of the opening comments should include:

   - the overall purpose of the software module,
   - the names of the programmers,
   - the creation (optional) and last update dates,
   - the hardware/software configuration required to use the module, and
   - any copyright information.
2. **Including .h files**

Nested includes in the header file should be avoided. Nested includes obscure the manner in which the modules are interconnected. The only exception is if data structures or functions depend on definitions made in other modules, such as `typedefs`. On the other hand, an implementation file can include other header and implementation files.

3. **#define statements**

Public constants and macros are next. Special care is required to determine if a definition should be made private or public. One approach to this question is to begin with everything defined as private, and then shift definitions into the public category only when deemed necessary for the client to access in order to use the module. If the parameter relates to what the module does or how to use the module, then it should probably be public. On the other hand, if it relates to how it works or how it is implemented, it should probably be private.

4. **struct, union, enum statements**

The definitions of public structures allow the client software to create data structures specific for this module.

5. **Global variables and constants**

If at all possible, public global variables should be avoided. Public constants follow the same rules as public definitions. If the client must have access to a constant to use the module, then it could be placed in the header file.

6. **Prototypes of public functions**

The prototypes for the public functions are last. Just like the implementation file, we will arrange the code implementations in a top-down fashion. Comments should be directed to the client, and these comments should clarify what the function does and how the function can be used.
Formatting

Formatting is a matter of personal preference, but the following section lists techniques that can make your software easier to understand, debug and change.

1. *Make the software easy to read*

   We should develop and debug software by observing it on the computer screen. In order to eliminate horizontal scrolling, no line of code should be more than 80 characters wide. If we do make hard copy printouts of the software at the end of a project, this rule will result in a printout that is easy to read.

2. *Indentation should be set at 2 spaces*

   When transporting code from one computer to another, the tab settings may be different. So, what looks good on one computer may look ugly on another. For this reason, we should avoid tabs and use just spaces. Local variable definitions can go on the same line as the function definition, or in the third column on the next line.

3. *Be consistent about where we put spaces*

   Similar to English punctuation, there should be no space before a comma or a semicolon, but there should be at least one space or a carriage return after a comma or a semicolon. There should be no space before or after open or close parentheses. Assignment and comparison operations should have a single space before and after the operation. One exception to the single space rule is if there are multiple assignment statements, we can line up the operators and values. For example

   ```
   Data     = 1;
   pressure = 100;
   voltage  = 5;
   ```
4. Be consistent about where we put braces {}

Misplaced braces cause both syntax and semantic errors, so it is critical to maintain a consistent style. Place the opening brace on a new line directly underneath the code that opens the scope of the compound statement. Placing the open brace at the beginning of a new line provides a visual clue that a new code block has started. Place the closing brace on a separate line to give a vertical separation showing the end of the compound statement. The horizontal placement of the close brace should line up with the opening brace, giving a visual clue that the enclosed code is a compound statement. For example

```c
void main(void)
{
  int i, j, k;
  j = 1;
  if (sub0(j))
  {
    for (i = 0; i < 6; i++)
    {
      sub1(i);
      k = sub2(i, j);
    }
  }
  else
  {
    k = sub3();
  }
}
```
Use braces after all \texttt{if-else}, \texttt{for}, \texttt{do-while}, \texttt{case} and \texttt{switch} commands where the following statement is \textit{not} a compound statement. For the case of single statements, it \textit{is} acceptable to leave out the braces, but we must be careful when editing and adding statements. For example, assume we start the following code:

\begin{verbatim}
if (flag)
    n = 0;
\end{verbatim}

Now, we add a second statement that we also want to execute if the flag is true. The following error might occur if we just add the new statement.

\begin{verbatim}
if (flag)
    n = 0;
    c = 0;
\end{verbatim}

We get the correct software if we enclose the two statements in braces:

\begin{verbatim}
if (flag)
{
    n = 0;
    c = 0;
}
\end{verbatim}

Leaving out braces for single statements increases our code density and is much more readable. For example:

\begin{verbatim}
if (flag)
    n = 0;
else
    n = 1;
\end{verbatim}

is better than:

\begin{verbatim}
if (flag)
{
    n = 0;
}
else
{
    n = 1;
}
\end{verbatim}
**Code Structure**

1. *Make the presentation easy to read*

We define presentation as the *look* and *feel* of our software as displayed on the screen. If at all possible, the size of our functions should be small enough so the majority of the code fits on a single computer screen. We must consider the presentation as a two-dimensional object. Consequently, we can reduce the 2-D area of our functions by encapsulating components and defining them as private functions.

Do not list multiple statements on the same line. The compiler often places debugging information on each line of code. Breakpoints in some systems can only be placed at the beginning of a line.

Consider the following two presentations. Since the compiler generates exactly the same code in each case, the computer execution will be identical. Therefore, we will focus on the differences in style.

The first example has a horrific style.

```c
void testFilter(short start, short stop, short step)
{
    short x, y; initFilter(); SCI_OutString("x(n) y(n)");
    SCI_OutChar(CR); for(x=start; x<=stop; x=x+step)
    { y=filter(x); SCI_OutUDec(x); SCI_OutChar(SP);
    SCI_OutUDec(y); SCI_OutChar(CR); } }
```

The second example places each statement on a separate line.

```c
void testFilter(short start, short stop, short step)
{
    short x, y;
    initFilter();
    SCI_OutString("x(n) y(n)");
    SCI_OutChar(CR);
    for (x = start; x <= stop; x += step)
    { y = filter(x);
    SCI_OutUDec(x);
    SCI_OutChar(SP);
    SCI_OutUDec(y);
    SCI_OutChar(CR); }
}
```
The "make the presentation easy to read" guideline sometimes comes into conflict with the "be consistent about where we put braces" guideline. For example, the following example is obviously easy to read, but violates the placement of braces rule.

\[
\text{for } (i = 0; i < 6; i++) \text{ dataBuf}[i] = 0;
\]

When in doubt, we will always be consistent with the braces (or lack of braces) rule. The correct style is also easy to read.

\[
\text{for } (i = 0; i < 6; i++)
\begin{align*}
dataBuf[i] &= 0;
\end{align*}
\]

2. **Employ modular programming techniques**

Complex functions should be broken into simple components, so that the details of the lower-level operations are hidden from the overall algorithms at the higher levels.

3. **Minimize scope**

In general, we hide the implementation of our software from its usage. The scope of a variable should be consistent with how the variable is used. In a military sense, we ask the question, "Which software has the need to know?" Global variables should be used only when the lifetime of the data is permanent, or when data needs to be passed from one thread to another. Otherwise, we should use local variables. When one module calls another, we should pass data using the normal parameter-passing mechanisms. As mentioned earlier, we consider I/O ports in a manner similar to global variables. There is no syntactic mechanism to prevent a module from accessing an I/O port, since the ports are at fixed and known absolute addresses. The Intel Pentium does have a complex hardware system to prevent unauthorized software from accessing I/O ports, but the details are beyond the scope of this document. For our embedded system, we must rely on the *does-access* rather than the *can-access* method. In other words, we must have the discipline to restrict I/O port access only in the module that is designed to access it. For similar reasons, we should consider each interrupt
service routine separately, grouping it with the corresponding I/O module. In 
particular, rather than having one long list of interrupt service routines for the 
entire system, each interrupt service routine should be separately defined 
along with the software that supports the other I/O hardware of the module. 
For example, the serial port interrupt service routine should be specified in 
the same file that handles setting up and using the serial port.

4. **Use types**

Using a `typedef` will clarify the format of a variable. It is another example 
of the separation of mechanism and policy. New data types and structures 
will begin with an upper case letter. The `typedef` allows us to hide the 
representation of the object and use an abstract concept instead. For example

```c
typedef short Temperature;
void main(void)
{
    Temperature lowT, highT;
}
```

This allows us to change the representation of temperature without having to 
find all the temperature variables in our software. Not every data type 
requires a `typedef`. We will use types for those objects of fundamental 
importance to our software, and for those objects for which a change in 
implementation is anticipated. As always, the goal is to clarify. If it doesn't 
make it easier to understand, easier to debug, or easier to change, don't do it.

5. **Prototype all functions**

Public functions obviously require a prototype in the header file. In the 
implementation file, we will organize the software in a top-down hierarchical 
fashion. Since the highest level functions go first, prototypes for the lower-
level private functions will be required. Grouping the lowlevel prototypes at 
the top provides a summary overview of the software in this module. Include 
both the type and name of the input parameters. Specify the function as 
(\texttt{void}) if it has no parameters.
If the implementation is not naturally hierarchical, then we will order the functions in normal order of use. For example, start with the initialization functions, followed by the operations, and end with the shutdown functions. For example:

```
SCI_Init();
SCI_Out();
SCI_In();
SCI_Shutdown();
```

These prototypes are easy to understand:

```
void start(unsigned short period, void(*functionPt)(void));
short divide(short dividend, short divisor);
void SCI_Init(void);
```

These prototypes are harder to understand:

```
start(unsigned short, (*))();
short divide(short, short);
SCI_Init();
```

6. **Declare function return types explicitly**

In general, we can remove ambiguities by clarifying exactly what we want. Unless the number of parameters is large, we will place the return type, the function name, and the input parameters on a single line. The following are good examples of the first line of several functions.

```
void main(void)
void SCI_OutUDec(unsigned short number)
unsigned short SCI_InUHex(void)
int RxFIFO_Put(char data)
```

7. **Declare data and parameters as const whenever possible**

Declaring an object as `const` has two advantages. The compiler can produce more efficient code when dealing with parameters that don't change. The second advantage is to catch software bugs, i.e., situations where the program incorrectly attempts to modify data that it should not modify.
8. **goto statements are not allowed**

Debugging is hard enough without adding the complexity generated when using `goto`. When developing assembly language software, we should restrict the branching operations to the simple structures allowed in C.

9. **++ and -- should appear once in complex statements**

These operations should only appear as commands by themselves. Also, the compiler tends to generate more efficient code when they are separated. More importantly, the issue is readability. The statement:

```
*(-pt) = buffer[n++];
```

should be written as:

```
--pt;
*(pt) = buffer[n++];
```

or as:

```
*(-pt) = buffer[n];
n++;
```

10. **Be a parenthesis zealot**

When mixing arithmetic, logical, and conditional operations, explicitly specify the order of operations. Do not rely on the order of precedence. As always, the major issue is clarity. Even if the following code were actually to perform the intended operation (which in fact it does not),

```
if ( x + 1 & 0x0F == y | 0x04)
```

the programmer assigned to modify it in the future will have a better chance if we had written:

```
if (((x + 1) & 0x0F) == (y | 0x04))
```
11. *Use enum instead of #define or const.*

The use of `enum` allows for consistency checking during compilation, and provides for easy to read software. A good optimizing compiler will create the same object code for the following four examples. So once again, we focus on style.

In the first example, we need comments to explain the operations:

```c
void function1(void)
{
    Mode = 1; // no error
}

void function2(void)
{
    if (Mode == 0)
        // error?
        SCI_OutString("error");
    Mode = 3;   // This line will compile
    Mode = 256; // This line will compile
}
```

In the second example, no comments are needed:

```c
#define NOERROR 1
#define ERROR 0

int Mode;

void function1(void)
{
    Mode = NOERROR;
}

void function2(void)
{
    if (Mode == ERROR)
        SCI_OutString("error");
    Mode = 3;   // This line will compile
    Mode = 256; // This line will compile
}
```
In the third example, the compiler performs a type-match, making sure `Mode`, `NOERROR`, and `ERROR` are the same type:

```c
const unsigned char NOERROR = 1;
const unsigned char ERROR = 0;

unsigned char Mode;

void function1(void)
{
    Mode = NOERROR;
}

void function2(void)
{
    if (Mode == ERROR)
        SCI_OutString("error");
    Mode = 3; // This will compile
    Mode = 256; // This line will *NOT* compile (Mode>255)
}
```

 Enumeration provides a check of both type and value, if the compiler supports it. Standard C compilers do **NOT** support it, but C++ compilers do. However, in C it is good programming practice to see the type required by a variable or function parameter, and provide the necessary enumerated type. We can explicitly set the values of the enumerated types if needed.

```c
enum ModeState {ERROR, NOERROR};
enum ModeState Mode;

void function1(void)
{
    Mode = NOERROR;
}

void function2(void)
{
    if (Mode == ERROR)
        SCI_outString("error");
    Mode = 3; // This will *NOT* compile in C++ (out of range)
    Mode = 256; // This line will *NOT* compile (out of range)
}
```
12. Don't use bit-shift for arithmetic operations

Microcomputer architectures and compilers used to be so limited that it made sense to perform multiply/divide by 2 using a shift operation. For example, when multiplying a number by 4, we might be tempted to write \( \text{data} \ll 2 \). This is wrong; if the operation is multiply, we should write \( \text{data} \times 4 \). Compiler optimization has developed to the point where the compiler can choose to implement \( \text{data} \times 4 \) as either a shift or multiply depending on the instruction set of the computer. When we use \( \text{data} \times 4 \), we have code that is easier to understand than \( \text{data} \ll 2 \).
Comments

Comments are the least important aspect involved in writing quality software. Well-organized software with simple interfaces having operations so easy to understand makes comments unnecessary.

The beginning of every file should include the file name, purpose, hardware connections, programmer, date, and copyright. For example,

```c
// ----------------------------------------
// Filename:
// SCI.h
// Description:
// I/O routines for MC9S12 serial communication interface
// Author:
// PMcL
// Last modified:
// 8-Mar-06
// Copyright (c) 2006 by Peter McLean
```

The beginning of every function should include a line delimiting the start of the function, the purpose, input parameters, output parameters, and special conditions that apply. The comments at the beginning of the function explain the policies (e.g., how to use the function.) These comments, which are similar to the comments for the prototypes in the header file, are intended to be read by the client. For example,

```c
//SCI_Init
//Initialize the Serial Communication Interface
//Input:
//baudRate is the baud rate in bits/sec
//Output:
//none
//Conditions:
//Assumes a module clock frequency of 24 MHz
//SCIBDL = 1500000/baudRate
//e.g. baudRate = 115200 bits/sec, SCIBDL=13
```
Comments can be added to a variable or constant definition to clarify the usage. In particular, comments can specify the units of the variable or constant. For complicated situations, we can use additional lines and include examples. For example,

```c
short V1;              // voltage at node 1 in mV,
                       // range -5000 mV to +5000 mV

unsigned short Fs;    // sampling rate in Hz

int FoundFlag;        // 0 if keyword not yet found,
                       // 1 if found

enum TMode           // system states for the serial port
{
    IDLE,
    RECEIVE,
    TRANSMIT
};

enum TMode Mode;      // determines serial port action
```

Comments can be used to describe complex algorithms. These types of comments are intended to be read by our coworkers. The purpose of these comments is to assist in changing the code in the future, or applying this code to a similar but slightly different application. Comments that restate the function provide no additional information, and actually make the code harder to read. Examples of bad comments include:

```c
time++;      // add one to time
mode = IDLE; // set mode to IDLE
```

Good comments explain why the operation is performed, and what it means:

```c
time++;      // maintain elapsed time in msec
mode = IDLE; // switch to idle mode
            // because no more data is available
```

We can add spaces so the comment fields line up. As stated earlier, we avoid tabs because they often do not translate from one system to another. In this way, the software is on the left and the comments can be read on the right.
Alternatively, comments can appear on the line before the actual code: It is also good practice to separate blocks of code by blank lines.

```c
void main(void)
{
    // Initialise ModCon
    ModCon_Init();

    // Loop forever (embedded software never ends!)
    while (1)
    {
        // Debug pulse on entry
        if (~DEBUG)
        {
            PTT |= 0x80;
        }

        // Receive commands from the PC
        ModCon_ReceiveFromPC();

        // Send status to the PC
        ModCon_SendToPC();

        // Update motors in background
        Motors_Update();

        // Debug pulse on exit
        if (~DEBUG)
        {
            PTT &= ~0x80;
        }
    }
}
```
Compiler Specific Coding Style

The CodeWarrior compiler’s default settings will generate a warning on many instances of correct code. Rather than turning off the warnings, which are designed to catch our programming errors, we will adopt a coding style which circumvents the creation of these warnings.

Warning : C1420: Result of function-call is ignored

This warning occurs because we should be using the result of a function-call. If we do not wish to use the result of a function call (for example, a function returns an error number that we don’t wish to handle), then we can adopt one of two strategies.

The first strategy is to typecast the function result to `void`. For example:

```c
void main(void)
{
    // Initialise ModCon
    (void)ModCon_Init();
    ...
```

The second strategy is to set up your code structure in a form where you can easily handle the expected result:

```c
void main(void)
{
    // Initialise ModCon
    if (!ModCon_Init())
    {
        // Error handling goes here
    };
    ...
```
C4000: Condition always TRUE

This warning occurs when we implement our “loop forever” \texttt{while} loop with an argument which is constant:

```c
void main(void)
{
    // Loop forever (embedded software never ends!)
    while (1)
    {
        // Do our stuff...
    }
}
```

The way around this is to implement a \texttt{for} loop with no initialization, condition and post-processing parts:

```c
void main(void)
{
    // Loop forever (embedded software never ends!)
    for (;;) {
        // Do our stuff...
    }
}
```
ModCon Schematics
Analog Interface Schematics
Differential Amplifiers for ADC Input

Overvoltage Protection

Analog to Digital Converter
Human-Machine Interface Schematics